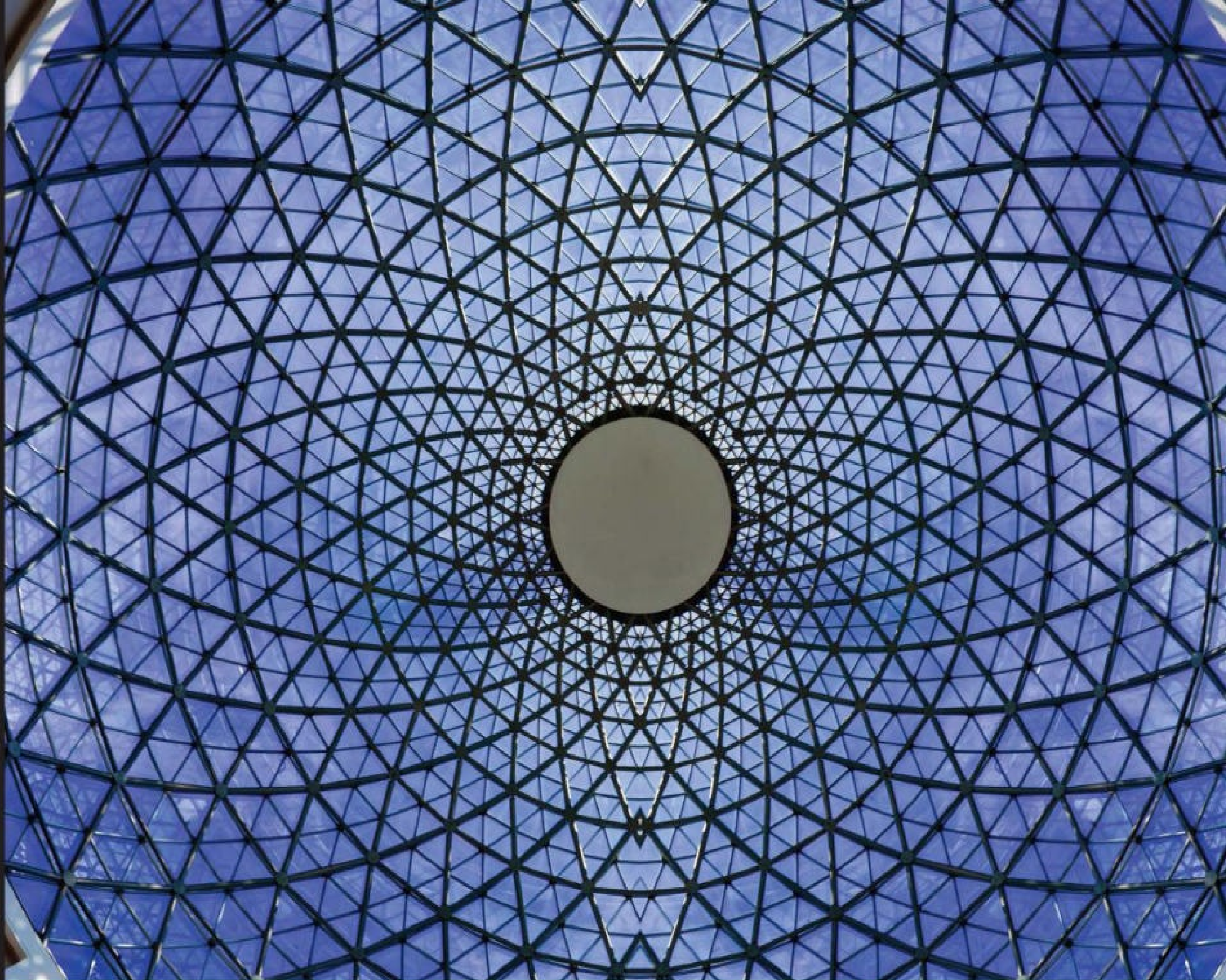


Quick answers to common problems

scikit-learn Cookbook

Over 50 recipes to incorporate scikit-learn into every step of the data science pipeline, from feature extraction to model building and model evaluation

Trent Hauck

[PACKT] PUBLISHING open source* community experience distilled

目錄

Scikit-learn 秘籍

> 原书：Scikit-learn Cookbook

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

译者

	章节	译者
1	预处理	muxuezi
2	回归	muxuezi
3	聚类	飞龙
4	分类	飞龙
5	后处理	飞龙

协议

第一章 模型预处理

> 作者：Trent Hauck
>
> 译者：muxuezi
>
>

本章包括以下主题：

简介

本章介绍数据获取（setting data），数据整理（preparing data）和建模前的降维（premodel dimensionality reduction）工作。这些内容并非机器学习（machine learning，ML）最核心的部分，但是它们往往决定模型的成败。

本章主要分三部分。首先，我们介绍如何创建模拟数据（fake data），这看着微不足道，但是创建模拟数据并用模型进行拟合是模型测试的重要步骤。更重要的是，当我们从零开始一行一行代码实现一个算法时，我们想知道算法功能是否达到预期，这时手上可能没有数据，我们可以创建模拟数据来测试。之后，我们将介绍一些数据预处理变换的方法，包括缺失数据填补（data imputation），分类变量编码（categorical variable encoding）。最后，我们介绍一些降维方法，如主成分分析，因子分析，以及正态随机过程等。

本章，尤其是前半部分与后面的章节衔接紧密。后面使用scikit-learn时，数据都源自本章内容。前两节介绍数据获取；紧接着介绍数据清洗。

> 本书使用scikit-learn 0.15，NumPy 1.9和pandas 0.13，兼容Python2.7和Python3.4。还会用到其他的Python库，建议参考对应的官方安装指令。

1.1 从外部源获取样本数据

如果条件允许，学本书内容时尽量用你熟悉的数据集；方便起见，我们用scikit-learn的内置数据库。这些内置数据库可用于测试不同的建模技术，如回归和分类。而且这些内置数据库都是非常著名的数据库。这对不同领域的学术论文的作者们来说是很用的，他们可以用这些内置数据库将他们的模型与其他模型进行比较。

> 推荐使用IPython来运行文中的指令。大内存很重要，这样可以让普通的命令正常运行。如果用IPython Notebook就更好了。如果你用Notebook，记得用 `%matplotlib inline` 指令，这样图象就会出现在Notebook里面，而不是一个新窗口里。

Getting ready

scikit-learn的内置数据库在 `datasets` 模块里。用如下命令导入：

```
from sklearn import datasets
import numpy as np
```

在IPython里面运行 `datasets.*?` 就会看到 `datasets` 模块的指令列表。

How to do it...

`datasets` 模块主要有两种数据类型。较小的测试数据集在 `sklearn` 包里面，可以通过 `datasets.load_*?` 查看。较大的数据集可以根据需要下载。后者默认情况下不在 `sklearn` 包里面；但是，有时这些大数据集可以更好的测试模型和算法，因为比较复杂足以模拟现实情形。

默认在 `sklearn` 包里面的数据集可以通过 `datasets.load_*?` 查看。另外一些数据集需要通过 `datasets.fetch_*?` 下载，这些数据集更大，没有被自动安装。经常用于测试那些解决实际问题的算法。

首先，加载 `boston` 数据集看看：

```
boston = datasets.load_boston()
print(boston.DESCR)
```

```
Boston House Prices dataset

Notes
------
Data Set Characteristics:
```

```
    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots
 over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per t
own
        - CHAS     Charles River dummy variable (= 1 if tract bo
unds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 mil
lion)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prio
r to 1940
        - DIS      weighted distances to five Boston employment
centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion
of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000
's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
http://archive.ics.uci.edu/ml/datasets/Housing


This dataset was taken from the StatLib library which is maintai
ned at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L.
'Hedonic
prices and the demand for clean air', J. Environ. Economics & Ma
nagement,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regressio
n diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in th
e table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learni
ng papers that address regression
```

```
problems.

**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying
 Influential Data and Sources of Collinearity', Wiley, 1980. 244
-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based
 Learning. In Proceedings on the Tenth International Conference
of Machine Learning, 236-243, University of Massachusetts, Amher
st. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Hous
ing)
```

`DESCR` 将列出数据集的一些概况。下面我们来下载一个数据集：

```
housing = datasets.fetch_california_housing()
print(housing.DESCR)
```

```
downloading Cal. housing from http://lib.stat.cmu.edu/modules.ph
p?op=modload&name=Downloads&file=index&req=getit&lid=83 to C:\Us
ers\tj2\scikit_learn_data
California housing dataset.

The original database is available from StatLib

    http://lib.stat.cmu.edu/

The data contains 20,640 observations on 9 variables.

This dataset contains the average house value as target variable
and the following input variables (features): average income,
housing average age, average rooms, average bedrooms, population
,
average occupation, latitude, and longitude in that order.

References
----------

Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions
,
Statistics and Probability Letters, 33 (1997) 291-297.
```

How it works...

当这些数据集被加载时，它们并不是直接转换成Numpy数组。它们是 `Bunch` 类型。**Bunch**是Python常用的数据结构。基本可以看成是一个词典，它的键被实例对象作为属性使用。

用 `data` 属性连接数据中包含自变量的Numpy数组，用 `target` 属性连接数据中的因变量。

```
X, y = boston.data, boston.target
```

网络上 `Bunch` 对象有不同的实现；自己写一个也不难。scikit-learn用基本模块定义 `Bunch` 。

There's more...

让你从外部源获取数据集时，它默认会被当前文件夹的 `scikit_learn_data/` 放在里面，可以通过两种方式进行配置：

- 设置 `SCIKIT_LEARN_DATA` 环境变量指定下载位置
- `fetch_*?` 方法的第一个参数是 `data_home` ，可以知道下载位置

通过 `datasets.get_data_home()` 很容易检查默认下载位置。

See also

UCI机器学习库（UCI Machine Learning Repository）是找简单数据集的好地方。很多scikit-learn的数据集都在那里，那里还有更多的数据集。其他数据源还是著名的KDD和Kaggle。

1.2 创建试验样本数据

希望你在学习本书时用自己的数据来试验，如果实在没有数据，下面就介绍如何用scikit-learn创建一些试验用的样本数据（toy data）。

Getting ready

与前面获取内置数据集，获取新数据集的过程类似，创建样本数据集，用 `make_数据集名称` 函数。这些数据集都是人造的：

```
from sklearn import datasets

datasets.make_*?
```

```
datasets.make_biclusters
datasets.make_blobs
datasets.make_checkerboard
datasets.make_circles
datasets.make_classification
datasets.make_friedman1
datasets.make_friedman2
datasets.make_friedman3
datasets.make_gaussian_quantiles
datasets.make_hastie_10_2
datasets.make_low_rank_matrix
datasets.make_moons
datasets.make_multilabel_classification
datasets.make_regression
datasets.make_s_curve
datasets.make_sparse_coded_signal
datasets.make_sparse_spd_matrix
datasets.make_sparse_uncorrelated
datasets.make_spd_matrix
datasets.make_swiss_roll
```

为了简便，下面我们用 `d` 表示 `datasets` ， `np` 表示 `numpy` ：

```
import sklearn.datasets as d
import numpy as np
```

How to do it...

这一节将带你创建几个数据集；在后面的*How it works...*一节，我们会检验这些数据集的特性。除了样本数据集，后面还会创建一些具有特定属性的数据集来显示算法的特点。

首先，我们创建回归（regression）数据集：

```
reg_data = d.make_regression()
reg_data[0].shape,reg_data[1].shape
```

```
((100, 100), (100,))
```

`reg_data` 默认是一个元组，第一个元素是100×100的矩阵——100个样本，每个样本10个特征（自变量），第二个元素是1个因变量，对应自变量的样本数量，也是100个样本。然而，默认情况下，只有10个特征与因变量的相关（参数 `n_informative` 默认值是10），其他90个特征都与。

可以自定义更复杂的数据集。比如，创建一个1000×10的矩阵，5个特征与因变量相关，误差系数0.2，两个因变量。代码如下所示：

```
complex_reg_data = d.make_regression(1000, 10, 5, 2, 1.0)
complex_reg_data[0].shape,complex_reg_data[1].shape
```

```
((1000, 10), (1000, 2))
```

分类数据集也很容易创建。很容易创建一个基本均衡分类集，但是这种情况现实中几乎不可能发生——大多数用户不会改变消费习惯，大多数交易都不是虚假的，等等。因此，创建一个非均衡数据集更有意义：

```
classification_set = d.make_classification(weights=[0.1])
np.bincount(classification_set[1])
```

```
array([10, 90], dtype=int64)
```

聚类数据集也可以创建。有一些函数可以为不同聚类算法创建对应的数据集。例如，`blobs` 函数可以轻松创建K-Means聚类数据集：

```
%matplotlib inline
import sklearn.datasets as d
from matplotlib import pyplot as plt
import numpy as np

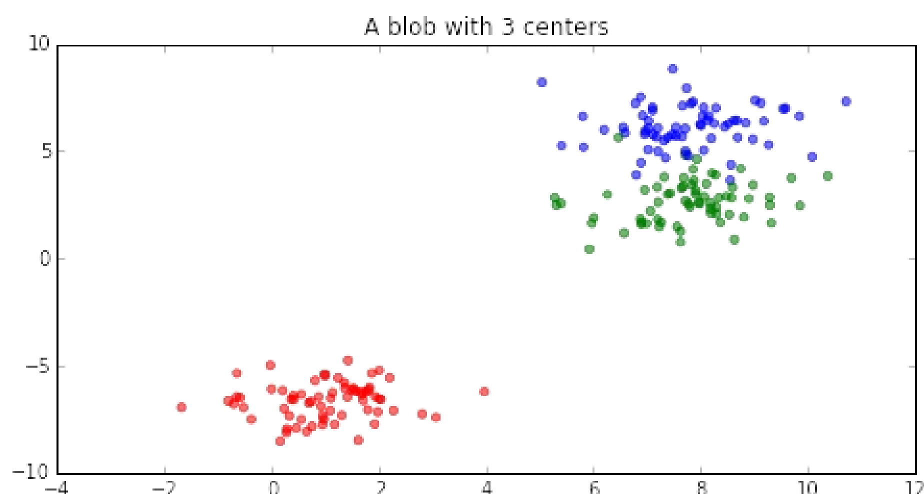
blobs = d.make_blobs(200)

f = plt.figure(figsize=(8, 4))

ax = f.add_subplot(111)
ax.set_title("A blob with 3 centers")

colors = np.array(['r', 'g', 'b'])
ax.scatter(blobs[0][:, 0], blobs[0][:, 1], color=colors[blobs[1]
.astype(int)], alpha=0.75)
```

```
<matplotlib.collections.PathCollection at 0x88e44e0>
```



How it works...

下面让我们从源代码看看scikit-learn是如何生成回归数据集的。下面任何未重新定义的参数都使用 `make_regression` 函数的默认值。

其实非常简单。首先，函数调用时生成一个指定维度的随机数组。

```
X = np.random.randn(n_samples, n_features)
```

对于基本均衡数据集，其目标数据集生成方法是：

```
ground_truth = np.zeroes((np_samples, n_target))
ground_truth[:n_informative, :] = 100*np.random.rand(n_informati
ve,
n_targets)
```

然后 `X` 和 `ground_truth` 点积加上 `bias` 就得到了 `y` ：

```
y = np.dot(X, ground_truth) + bias
```

> 点积是一种基本的矩阵运算$A_{m \times n} \cdot B_{n \times s} = C_{m \times s}$。因此， `y` 数据集里面样本数量是 `n_samples` ，即数据集的行数，因变量数量是 `n_target` 。

由于Numpy的传播操作（broadcasting）， `bias` 虽然是标量，也会被增加到矩阵的每个元素上。增加噪声和数据混洗都很简单。这样试验用的回归数据集就完美了。

1.3 把数据调整为标准正态分布

经常需要将数据标准化调整（scaling）为标准正态分布（standard normal）。标准正态分布算得上是统计学中最重要的分布了。如果你学过统计，Z值表（z-scores）应该不陌生。实际上，Z值表的作用就是把服从某种分布的特征转换成标准正态分布的Z值。

Getting ready

数据标准化调整是非常有用的。许多机器学习算法在具有不同范围特征的数据中呈现不同的学习效果。例如，SVM（Support Vector Machine，支持向量机）在没有标准化调整过的数据中表现很差，因为可能一个变量的范围是0-10000，而另一个变量的范围是0-1。`preprocessing` 模块提供了一些函数可以将特征调整为标准形：

```
from sklearn import preprocessing
import numpy as np
```

How to do it...

还用 `boston` 数据集运行下面的代码：

```
from sklearn import datasets
boston = datasets.load_boston()
X, y = boston.data, boston.target
```

```
X[:, :3].mean(axis=0) #前三个特征的均值
```

```
array([  3.59376071,  11.36363636,  11.13677866])
```

```
X[:, :3].std(axis=0) #前三个特征的标准差
```

```
array([  8.58828355,  23.29939569,   6.85357058])
```

这里看出很多信息。首先，第一个特征的均值是三个特征中最小的，而其标准差却比第三个特征的标准差大。第二个特征的均值和标准差都是最大的——说明它的值很分散，我们通过 `preprocessing` 对它们标准化：

```
X_2 = preprocessing.scale(X[:, :3])
```

```
X_2.mean(axis=0)
```

```
array([  6.34099712e-17,  -6.34319123e-16,  -2.68291099e-15])
```

```
X_2.std(axis=0)
```

```
array([ 1.,  1.,  1.])
```

How it works...

中心化与标准化函数很简单，就是减去均值后除以标准差，公式如下所示：

$$x= \frac {x- \bar x} \sigma$$

除了这个函数，还有一个中心化与标准化类，与管线命令（Pipeline）联合处理大数据集时很有用。单独使用一个中心化与标准化类实例也是有用处的：

```
my_scaler = preprocessing.StandardScaler()
my_scaler.fit(X[:, :3])
my_scaler.transform(X[:, :3]).mean(axis=0)
```

```
array([  6.34099712e-17,  -6.34319123e-16,  -2.68291099e-15])
```

把特征的样本均值变成 `0` ，标准差变成 `1` ，这种标准化处理并不是唯一的方法。 `preprocessing` 还有 `MinMaxScaler` 类，将样本数据根据最大值和最小值调整到一个区间内：

```
my_minmax_scaler = preprocessing.MinMaxScaler()
my_minmax_scaler.fit(X[:, :3])
my_minmax_scaler.transform(X[:, :3]).max(axis=0)
```

```
array([ 1.,  1.,  1.])
```

通过 `MinMaxScaler` 类可以很容易将默认的区间 `0` 到 `1` 修改为需要的区间：

```
my_odd_scaler = preprocessing.MinMaxScaler(feature_range=(-3.14,
3.14))
my_odd_scaler.fit(X[:, :3])
my_odd_scaler.transform(X[:, :3]).max(axis=0)
```

```
array([ 3.14,  3.14,  3.14])
```

还有一种方法是正态化（normalization）。它会将每个样本长度标准化为1。这种方法和前面介绍的不同，它的特征值是标量。正态化代码如下：

```
normalized_X = preprocessing.normalize(X[:, :3])
```

乍看好像没什么用，但是在求欧式距离（相似度度量指标）时就很必要了。例如三个样本分别是向量 `(1,1,0)`，`(3,3,0)`，`(1,-1,0)`。样本1与样本3的距离比样本1与样本2的距离短，尽管样本1与样本3是轴对称，而样本1与样本2只是比例不同而已。由于距离常用于相似度检测，因此建模之前如果不对数据进行正态化很可能会造成失误。

There's more...

数据填补（data imputation）是一个内涵丰富的主题，在使用scikit-learn的数据填补功能时需要注意以下两点。

创建幂等标准化（**idempotent scaler**）对象

有时可能需要标准化 `StandardScaler` 实例的均值和/或方差。例如，可能（尽管没用）会经过一系列变化创建出一个与原来完全相同的 `StandardScaler`：

```
my_useless_scaler = preprocessing.StandardScaler(with_mean=False
, with_std=False)
transformed_sd = my_useless_scaler.fit_transform(X[:, :3]).std(a
xis=0)
original_sd = X[:, :3].std(axis=0)
np.array_equal(transformed_sd, original_sd)
```

```
True
```

处理稀疏数据填补

在标准化处理时，稀疏矩阵的处理方式与正常矩阵没太大不同。这是因为数据经过中心化处理后，原来的 `0` 值会变成非 `0` 值，这样稀疏矩阵经过处理就不再稀疏了：

```
import scipy
matrix = scipy.sparse.eye(1000)
preprocessing.scale(matrix)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-45-466df6030461> in <module>()
      1 import scipy
      2 matrix = scipy.sparse.eye(1000)
----> 3 preprocessing.scale(matrix)


d:\programfiles\Miniconda3\lib\site-packages\sklearn\preprocessing\data.py in scale(X, axis, with_mean, with_std, copy)
    120         if with_mean:
    121             raise ValueError(
--> 122                 "Cannot center sparse matrices: pass `with_mean=False` instead"
    123                 " See docstring for motivation and alternatives.")
    124         if axis != 0:


ValueError: Cannot center sparse matrices: pass `with_mean=False` instead See docstring for motivation and alternatives.
```

这个错误表面，标准化一个稀疏矩阵不能带 `with_mean`，只要 `with_std`：

```
preprocessing.scale(matrix, with_mean=False)
```

```
<1000x1000 sparse matrix of type '<class 'numpy.float64'>'
    with 1000 stored elements in Compressed Sparse Row format>
```

另一个方法是直接处理 `matrix.todense()`。但是，这个方法很危险，因为矩阵已经是稀疏的了，这么做可能出现内存异常。

1.4 用阈值创建二元特征

在前一个主题，我们介绍了数据转换成标准正态分布的方法。现在，我们看看另一种完全不同的转换方法。

当不需要呈标准化分布的数据时，我们可以不处理它们直接使用；但是，如果有足够理由，直接使用也许是聪明的做法。通常，尤其是处理连续数据时，可以通过建立二元特征来分割数据。

Getting ready

通常建立二元特征是非常有用的方法，不过要格外小心。我们还是用 `boston` 数据集来学习如何创建二元特征。

首先，加载 `boston` 数据集：

```
from sklearn import datasets
boston = datasets.load_boston()
import numpy as np
```

How to do it...

与标准化处理类似，scikit-learn有两种方法二元特征：

- `preprocessing.binarize` （一个函数）
- `preprocessing.Binarizer` （一个类）

`boston` 数据集的因变量是房子的价格中位数（单位：千美元）。这个数据集适合测试回归和其他连续型预测算法，但是假如现在我们想预测一座房子的价格是否高于总体均值。要解决这个问题，我们需要创建一个均值的阈值。如果一个值比均值大，则为 `1` ；否则，则为 `0` ：

```
from sklearn import preprocessing
new_target = preprocessing.binarize(boston.target, threshold=bos
ton.target.mean())
new_target[0,:5]
```

```
array([ 1.,  0.,  1.,  1.,  1.])
```

很容易，让我们检查一下：

```
(boston.target[:5] > boston.target.mean()).astype(int)
```

```
array([1, 0, 1, 1, 1])
```

既然Numpy已经很简单了，为什么还要用scikit-learn的函数呢？管道命令，将在用管道命令联接多个预处理步骤一节中介绍，会解释这个问题；要用管道命令就要用 `Binarizer` 类：

```
bin = preprocessing.Binarizer(boston.target.mean())
new_target = bin.fit_transform(boston.target)
new_target[0,:5]
```

```
array([ 1.,  0.,  1.,  1.,  1.])
```

How it works...

方法看着非常简单；其实scikit-learn在底层创建一个检测层，如果被监测的值比阈值大就返回 `Ture` 。然后把满足条件的值更新为 `1` ，不满足条件的更新为 `0` 。

There's more...

让我们再介绍一些稀疏矩阵和 `fit` 方法的知识。

稀疏矩阵

稀疏矩阵的 `0` 是不被存储的；这样可以节省很多空间。这就为 `binarizer` 造成了问题，需要指定阈值参数 `threshold` 不小于 `0` 来解决，如果 `threshold` 小于 `0` 就会出现错误：

```
from scipy.sparse import coo
spar = coo.coo_matrix(np.random.binomial(1, .25, 100))
preprocessing.binarize(spar, threshold=-1)
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-31-c9b5156c63ab> in <module>()
      1 from scipy.sparse import coo
      2 spar = coo.coo_matrix(np.random.binomial(1, .25, 100))
----> 3 preprocessing.binarize(spar, threshold=-1)


d:\programfiles\Miniconda3\lib\site-packages\sklearn\preprocessing\data.py in binarize(X, threshold, copy)
    718     if sparse.issparse(X):
    719         if threshold < 0:
--> 720             raise ValueError('Cannot binarize a sparse matrix with threshold '
    721                              '< 0')
    722         cond = X.data > threshold


ValueError: Cannot binarize a sparse matrix with threshold < 0
```

`fit` 方法

`binarizer` 类里面有 `fit` 方法，但是它只是通用接口，并没有实际的拟合操作，仅返回对象。

1.5 分类变量处理

分类变量是经常遇到的问题。一方面它们提供了信息；另一方面，它们可能是文本形式——纯文字或者与文字相关的整数——就像表格的索引一样。

因此，我们在建模的时候往往需要将这些变量量化，但是仅仅用简单的 `id` 或者原来的形式是不行的。因为我们也需要避免在上一节里通过阈值创建二元特征遇到的问题。如果我们把数据看成是连续的，那么也必须解释成连续的。

Getting ready

这里 `boston` 数据集不适合演示。虽然它适合演示二元特征，但是用来创建分类变量不太合适。因此，这里用 `iris` 数据集演示。

解决问题之前先把问题描述清楚。假设有一个问题，其目标是预测花萼的宽度；那么花的种类就可能是一个有用的特征。

首先，让我们导入数据：

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

现在 `X` 和 `y` 都获得了对应的值，我们把它们放到一起：

```
import numpy as np
d = np.column_stack((X, y))
```

How to do it...

下面我们把花类型 `y` 对应那一列转换成分类特征：

```
from sklearn import preprocessing
text_encoder = preprocessing.OneHotEncoder()
text_encoder.fit_transform(d[:, -1:]).toarray()[:5]
```

```
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

How it works...

这里，编码器为每个分类变量创建了额外的特征，转变成一个稀疏矩阵。矩阵是这样定义的：每一行由0和1构成，对应的分类特征是1，其他都是0。用稀疏矩阵存储数据很合理。

`text_encoder` 是一个标准的scikit-learn模型，可以重复使用：

```
text_encoder.transform(np.ones((3, 1))).toarray()
```

```
array([[ 0.,  1.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  1.,  0.]])
```

There's more...

在scikit-learn和Python库中，还有一些方法可以创建分类变量。如果你更倾向于用scikit-learn，而且分类编码原则很简单，可以试试 `DictVectorizer` 。如果你需要处理更复杂的分类编码原则， `patsy` 是很好的选择。

DictVectorizer

`DictVectorizer` 可以将字符串转换成分类特征：

```
from sklearn.feature_extraction import DictVectorizer
dv = DictVectorizer()
my_dict = [{'species': iris.target_names[i]} for i in y]
dv.fit_transform(my_dict).toarray()[:5]
```

```
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

> Python的词典可以看成是一个稀疏矩阵，它们只包含非0值。

Pasty

`patsy` 是另一个分类变量编码的包。经常和 `StatsModels` 一起用， `patsy` 可以把一组字符串转换成一个矩阵。

> 这部分内容与 scikit-learn关系不大，跳过去也没关系。

例如，如果 `x` 和 `y` 都是字符串， `dm = patsy.design_matrix("x + y")` 将创建适当的列。如果不是， `C(x)` 将生成一个分类变量。

例如，初看 `iris.target` ，可以把它当做是一个连续变量。因此，用下面的命令处理：

```
import patsy
patsy.dmatrix("0 + C(species)", {'species': iris.target})
```

```
DesignMatrix with shape (150, 3)
  C(species)[0]  C(species)[1]  C(species)[2]
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
              1              0              0
  [120 rows omitted]
  Terms:
    'C(species)' (columns 0:3)
  (to view full data, use np.asarray(this_obj))
```

1.6 标签特征二元化

在这个主题中，我们将用另一种方式来演示分类变量。有些时候只有一两个分类特征是重要的，这时就要避免多余的维度，如果有多个分类变量就有可能会出现这些多余的维度。

Getting ready

处理分类变量还有另一种方法，不需要通过 `OneHotEncoder` ，我们可以用 `LabelBinarizer` 。这是一个阈值与分类变量组合的方法。演示其用法之前，让我们加载 `iris` 数据集：

```
from sklearn import datasets as d
iris = d.load_iris()
target = iris.target
```

How to do it...

导入 `LabelBinarizer()` 创建一个对象：

```
from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer()
```

现在，将因变量的值转换成一个新的特征向量：

```
new_target = label_binarizer.fit_transform(target)
```

让我们看看 `new_target` 和 `label_binarizer` 对象的结果：

```
new_target.shape
```

```
(150, 3)
```

```
new_target[:5]
```

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       [1, 0, 0]])
```

```
new_target[-5:]
```

```
array([[0, 0, 1],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1]])
```

```
label_binarizer.classes_
```

```
array([0, 1, 2])
```

How it works...

`iris`的因变量基数为`3`，就是说有三种值。当`LabelBinarizer`将$N \times 1$向量转换成$N \times C$矩阵时，C就是$N \times 1$向量的基数。需要注意的是，当`label_binarizer`处理因变量之后，再转换基数以外的值都是`[0,0,0]`：

```
label_binarizer.transform([4])
```

```
array([[0, 0, 0]])
```

There's more...

0和1并不一定都是表示因变量中的阳性和阴性实例。例如，如果我们需要用`1000`表示阳性值，用`-1000`表示阴性值，我们可以用`label_binarizer`处理：

```
label_binarizer = LabelBinarizer(neg_label=-1000, pos_label=1000
)
label_binarizer.fit_transform(target)[:5]
```

```
array([[ 1000, -1000, -1000],
       [ 1000, -1000, -1000],
       [ 1000, -1000, -1000],
       [ 1000, -1000, -1000],
       [ 1000, -1000, -1000]])
```

> 阳性和阴性值的唯一限制是，它们必须为整数。

1.7 处理缺失值

实践中数值计算不可或缺，好在有很多方法可用，这个主题将介绍其中一些。不过，这些方法未必能解决你的问题。

scikit-learn有一些常见的计算方法，它可以对现有数据进行变换填补 `NA` 值。但是，如果数据集中的缺失值是有意而为之的——例如，服务器响应时间超过100ms——那么更合适的方法是用其他包解决，像处理贝叶斯问题的PyMC，处理风险模型的lifelines，或者自己设计一套方法。

Getting ready

处理缺失值的第一步是创建缺失值。Numpy可以很方便的实现：

```
from sklearn import datasets
import numpy as np
iris = datasets.load_iris()
iris_X = iris.data
masking_array = np.random.binomial(1, .25, iris_X.shape).astype(
bool)
iris_X[masking_array] = np.nan
```

让我们看看这几行代码，Numpy和平时用法不太一样，这里是在数组中用了一个数组作为索引。为了创建了随机的缺失值，先创建一个随机布尔值数组，其形状和 `iris_X` 数据集的维度相同。然后，根据布尔值数组分配缺失值。因为每次运行都是随机数据，所以 `masking_array` 每次都会不同。

```
masking_array[:5]
```

```
array([[False,  True, False, False],
       [False,  True, False, False],
       [False, False, False, False],
       [ True, False, False,  True],
       [False, False,  True, False]], dtype=bool)
```

```
iris_X[:5]
```

```
array([[ 5.1,  nan,  1.4,  0.2],
       [ 4.9,  nan,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ nan,  3.1,  1.5,  nan],
       [ 5. ,  3.6,  nan,  0.2]])
```

How to do it...

本书贯穿始终的一条原则（由于scikit-learn的存在）就是那些拟合与转换数据集的类都是可用的，可以在其他数据集中继续使用。具体演示如下所示：

```
from sklearn import preprocessing
impute = preprocessing.Imputer()
iris_X_prime = impute.fit_transform(iris_X)
iris_X_prime[:5]
```

```
array([[ 5.1       ,  3.05221239,  1.4       ,  0.2       ],
       [ 4.9       ,  3.05221239,  1.4       ,  0.2       ],
       [ 4.7       ,  3.2       ,  1.3       ,  0.2       ],
       [ 5.86306306,  3.1       ,  1.5       ,  1.21388889],
       [ 5.        ,  3.6       ,  3.82685185,  0.2       ]])
```

注意 `[3,0]` 位置的不同：

```
iris_X_prime[3,0]
```

```
5.863063063063064
```

```
iris_X[3,0]
```

```
nan
```

How it works...

上面的计算可以通过不同的方法实现。默认是均值 `mean` ，一共是三种：

- 均值 `mean` （默认方法）
- 中位数 `median`

- 众数 `most_frequent`

scikit-learn会用指定的方法计算数据集中的每个缺失值，然后把它们填充好。

例如，用 `median` 方法重新计算 `iris_X` ，重新初始化 `impute` 即可：

```
impute = preprocessing.Imputer(strategy='median')
iris_X_prime = impute.fit_transform(iris_X)
iris_X_prime[:5]
```

```
array([[ 5.1 ,  3.  ,  1.4 ,  0.2 ],
       [ 4.9 ,  3.  ,  1.4 ,  0.2 ],
       [ 4.7 ,  3.2 ,  1.3 ,  0.2 ],
       [ 5.8 ,  3.1 ,  1.5 ,  1.3 ],
       [ 5.  ,  3.6 ,  4.45,  0.2 ]])
```

如果数据有缺失值，后面计算过程中可能会出问题。例如，在*How to do it...*一节里面， `np.nan` 作为默认缺失值，但是缺失值有很多表现形式。有时用 `-1` 表示。为了处理这些缺失值，可以在方法中指定那些值是缺失值。方法默认缺失值表现形式是 `Nan` ，就是 `np.nan` 的值。

假设我们将 `iris_X` 的缺失值都用 `-1` 表示。看着很奇怪，但是 `iris` 数据集的度量值不可能是负数，因此用 `-1` 表示缺失值完全合理：

```
iris_X[np.isnan(iris_X)] = -1
iris_X[:5]
```

```
array([[ 5.1, -1. ,  1.4,  0.2],
       [ 4.9, -1. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [-1. ,  3.1,  1.5, -1. ],
       [ 5. ,  3.6, -1. ,  0.2]])
```

填充这些缺失值也很简单：

```
impute = preprocessing.Imputer(missing_values=-1)
iris_X_prime = impute.fit_transform(iris_X)
iris_X_prime[:5]
```

```
array([[ 5.1       ,  3.05221239,  1.4       ,  0.2       ],
       [ 4.9       ,  3.05221239,  1.4       ,  0.2       ],
       [ 4.7       ,  3.2       ,  1.3       ,  0.2       ],
       [ 5.86306306,  3.1       ,  1.5       ,  1.21388889],
       [ 5.        ,  3.6       ,  3.82685185,  0.2       ]])
```

There's more...

pandas库也可以处理缺失值，而且更加灵活，但是重用性较弱：

```
import pandas as pd
iris_X[masking_array] = np.nan
iris_df = pd.DataFrame(iris_X, columns=iris.feature_names)
iris_df.fillna(iris_df.mean())['sepal length (cm)'].head(5)
```

```
0    5.100000
1    4.900000
2    4.700000
3    5.863063
4    5.000000
Name: sepal length (cm), dtype: float64
```

其灵活性在于， `fillna` 可以填充任意统计参数值：

```
iris_df.fillna(iris_df.max())['sepal length (cm)'].head(5)
```

```
0    5.1
1    4.9
2    4.7
3    7.9
4    5.0
Name: sepal length (cm), dtype: float64
```

1.8 用管线命令处理多个步骤

管线命令不经常用，但是很有用。它们可以把多个步骤组合成一个对象执行。这样可以更方便灵活地调节和控制整个模型的配置，而不只是一个一个步骤调节。

Getting ready

这是我们把多个数据处理步骤组合成一个对象的第一部分。在scikit-learn里称为 `pipeline` 。这里我们首先通过计算处理缺失值；然后将数据集调整为均值为0，标准差为1的标准形。

让我们创建一个有缺失值的数据集，然后再演示 `pipeline` 的用法：

```
from sklearn import datasets
import numpy as np
mat = datasets.make_spd_matrix(10)
masking_array = np.random.binomial(1, .1, mat.shape).astype(bool
)
mat[masking_array] = np.nan
mat[:4, :4]
```

```
array([[ 1.05419595,  1.42287309,  0.02368264, -0.8505244 ],
       [ 1.42287309,  5.09704588,         nan, -2.46408728],
       [ 0.02368264,  0.03614203,  0.63317494,  0.09792298],
       [-0.8505244 , -2.46408728,  0.09792298,  2.04110849]])
```

How to do it...

如果不用管线命令，我们可能会这样实现：

```
from sklearn import preprocessing
impute = preprocessing.Imputer()
scaler = preprocessing.StandardScaler()
mat_imputed = impute.fit_transform(mat)
mat_imputed[:4, :4]
```

```
array([[ 1.05419595,  1.42287309,  0.02368264, -0.8505244 ],
       [ 1.42287309,  5.09704588,  0.09560571, -2.46408728],
       [ 0.02368264,  0.03614203,  0.63317494,  0.09792298],
       [-0.8505244 , -2.46408728,  0.09792298,  2.04110849]])
```

```
mat_imp_and_scaled = scaler.fit_transform(mat_imputed)
mat_imp_and_scaled[:4, :4]
```

```
array([[  1.09907483e+00,   2.62635324e-01,  -3.88958755e-01,
         -4.80451718e-01],
       [  1.63825210e+00,   2.01707858e+00,  -7.50508486e-17,
         -1.80311396e+00],
       [ -4.08014393e-01,  -3.99538476e-01,   2.90716556e+00,
          2.97005140e-01],
       [ -1.68651124e+00,  -1.59341549e+00,   1.25317595e-02,
          1.88986410e+00]])
```

现在我们用 `pipeline` 来演示：

```
from sklearn import pipeline
pipe = pipeline.Pipeline([('impute', impute), ('scaler', scaler)
])
```

我们看看 `pipe` 的内容。和前面介绍一致，管线命令定义了处理步骤：

```
pipe
```

```
Pipeline(steps=[('impute', Imputer(axis=0, copy=True, missing_va
lues='NaN', strategy='mean', verbose=0)), ('scaler', StandardSca
ler(copy=True, with_mean=True, with_std=True))])
```

然后在调用 `pipe` 的 `fit_transform` 方法，就可以把多个步骤组合成一个对象了：

```
new_mat = pipe.fit_transform(mat)
new_mat[:4, :4]
```

```
array([[  1.09907483e+00,   2.62635324e-01,  -3.88958755e-01,
         -4.80451718e-01],
       [  1.63825210e+00,   2.01707858e+00,  -7.50508486e-17,
         -1.80311396e+00],
       [ -4.08014393e-01,  -3.99538476e-01,   2.90716556e+00,
          2.97005140e-01],
       [ -1.68651124e+00,  -1.59341549e+00,   1.25317595e-02,
          1.88986410e+00]])
```

可以用Numpy验证一下结果：

```
np.array_equal(new_mat, mat_imp_and_scaled)
```

```
True
```

完全正确！本书后面的主题中，我们会进一步展示管线命令的威力。不仅可以用于预处理步骤中，在降维、算法拟合中也可以很方便的使用。

How it works...

前面曾经提到过，每个scikit-learn的算法接口都类似。`pipeline`最重要的函数也不外乎下面三个：

- `fit`
- `transform`
- `fit_transform`

具体来说，如果管线命令有`N`个对象，前`N-1`个对象必须实现`fit`和`transform`，第`N`个对象至少实现`fit`。否则就会出现错误。

如果这些条件满足，管线命令就会运行，但是不一定每个方法都可以。例如，`pipe`有个`inverse_transform`方法就是这样。因为由于计算步骤没有`inverse_transform`方法，一运行就有错误：

```
pipe.inverse_transform(new_mat)
```

```
---------------------------------------------------------------
-----------

AttributeError                            Traceback (most recent
 call last)

<ipython-input-12-62edd2667cae> in <module>()
----> 1 pipe.inverse_transform(new_mat)


d:\programfiles\Miniconda3\lib\site-packages\sklearn\utils\metae
stimators.py in <lambda>(*args, **kwargs)
     35             self.get_attribute(obj)
     36         # lambda, but not partial, allows help() to work
 with update_wrapper
---> 37         out = lambda *args, **kwargs: self.fn(obj, *args
, **kwargs)
     38         # update the docstring of the returned function
     39         update_wrapper(out, self.fn)


d:\programfiles\Miniconda3\lib\site-packages\sklearn\pipeline.py
 in inverse_transform(self, X)
    265         Xt = X
    266         for name, step in self.steps[::-1]:
--> 267             Xt = step.inverse_transform(Xt)
    268         return Xt
    269


AttributeError: 'Imputer' object has no attribute 'inverse_trans
form'
```

但是， `scalar` 对象可以正常运行：

```
scaler.inverse_transform(new_mat)[:4, :4]
```

```
array([[ 1.05419595,  1.42287309,  0.02368264, -0.8505244 ],
       [ 1.42287309,  5.09704588,  0.09560571, -2.46408728],
       [ 0.02368264,  0.03614203,  0.63317494,  0.09792298],
       [-0.8505244 , -2.46408728,  0.09792298,  2.04110849]])
```

只要把管线命令设置好，它就会如愿运行。它就是一组 `for` 循环，对每个步骤执行 `fit` 和 `transform` ，然后把结果传递到下一个变换操作中。

使用管线命令的理由主要有两点：

- 首先是方便。代码会简洁一些，不需要重复调用 `fit` 和 `transform` 。

- 其次，也是更重要的作用，就是使用交叉验证。模型可以变得很复杂。如果管线命令中的一个步骤调整了参数，那么它们必然需要重新测试；测试一个步骤参数的代码管理成本是很低的。但是，如果测试5个步骤的全部参数会变都很复杂。管线命令可以缓解这些负担。

1.9 用主成分分析降维

现在是时候升一级了！主成分分析（Principal component analysis，PCA）是本书介绍的第一个高级技术。到目前为止都是些简单的统计学知识，而PCA将统计学和线性代数组合起来实现降维，堪称简单模型的杀手锏。

Getting ready

PCA是scikit-learn的一个分解模块。还有一些分解模块后面会介绍。让我们用 `iris` 数据集演示一下，你也可以用自己的数据集：

```
from sklearn import datasets
iris = datasets.load_iris()
iris_X = iris.data
```

How to do it...

首先导入分解模块：

```
from sklearn import decomposition
```

然后，初始化一个PCA对象：

```
pca = decomposition.PCA()
pca
```

```
PCA(copy=True, n_components=None, whiten=False)
```

和scikit-learn其他对象相比，PCA的参数很少。这样PCA对象就创建了，下面用 `fit_transform` 处理 `iris_X` 数据：

```
iris_pca = pca.fit_transform(iris_X)
iris_pca[:5]
```

```
array([[ -2.68420713e+00,  -3.26607315e-01,   2.15118370e-02,
          1.00615724e-03],
       [ -2.71539062e+00,   1.69556848e-01,   2.03521425e-01,
          9.96024240e-02],
       [ -2.88981954e+00,   1.37345610e-01,  -2.47092410e-02,
          1.93045428e-02],
       [ -2.74643720e+00,   3.11124316e-01,  -3.76719753e-02,
         -7.59552741e-02],
       [ -2.72859298e+00,  -3.33924564e-01,  -9.62296998e-02,
         -6.31287327e-02]])
```

这样PCA就完成了，我们可以看看降维的效果：

```
pca.explained_variance_ratio_
```

```
array([ 0.92461621,  0.05301557,  0.01718514,  0.00518309])
```

How it works...

PCA是在数据分析中有一般性的数学定义和具体的应用场景。PCA用正交向量集表示原始数据集。

通常，PCA将原始数据集映射到新的空间中，里面每个列向量都是彼此正交的。从数据分析的视角看，PCA将数据集的协方差矩阵变换成若干能够“解释”一定比例变量的列向量。例如，在 `iris` 数据集中，92.5%的变量可以由第一个主成份表示。

数据分析里面维度多会导致维度灾难，因此降维至关重要。通常算法处理高维训练集时会出现拟合过度（overfit）的情况，于是难以把握测试集的一般性特征。如果数据集的真实结构可以用更少的维度表示，那么通常都值得一试。

为了演示这点，我们用PCA将 `iris` 数据集转换成二维数据。 `iris` 数据集用全部的维度通常可以很好的分类：

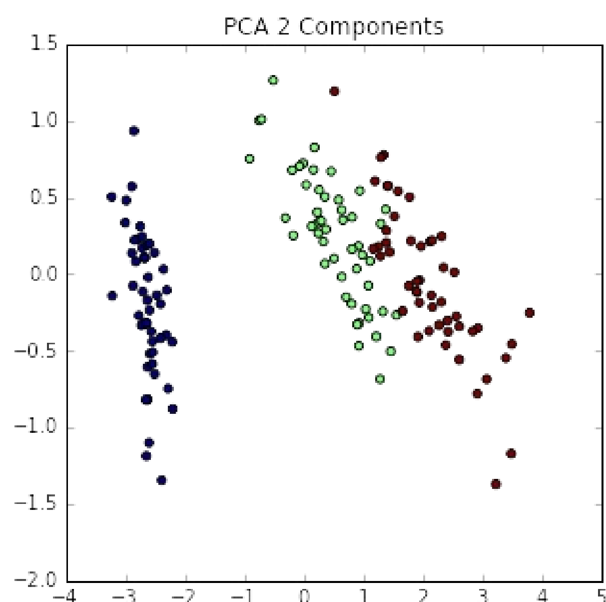
```
pca = decomposition.PCA(n_components=2)
iris_X_prime = pca.fit_transform(iris_X)
iris_X_prime.shape
```

```
(150, 2)
```

我们的矩阵现在是150×2，不是150×4了。二维变量更容易可视化：

```
%matplotlib inline
from matplotlib import pyplot as plt
f = plt.figure(figsize=(5, 5))
ax = f.add_subplot(111)
ax.scatter(iris_X_prime[:,0], iris_X_prime[:, 1], c=iris.target)
ax.set_title("PCA 2 Components")
```

```
<matplotlib.text.Text at 0x84571d0>
```



把数据集降成二维之后还是分离特征依然保留。我们可以查看这二维数据保留了多少变量信息：

```
pca.explained_variance_ratio_.sum()
```

```
0.97763177502480336
```

There's more...

PCA对象还可以一开始设置解释变量的比例。例如，如果我们想介绍98%的变量，PCA对象就可以这样创建：

```
pca = decomposition.PCA(n_components=.98)
iris_X_prime = pca.fit_transform(iris_X)
pca.explained_variance_ratio_.sum()
```

```
0.9948169145498101
```

```
iris_X_prime.shape
```

```
(150, 3)
```

由于我们想比二维主成份解释更多的变量，第三维就需要了。

1.10 用因子分析降维

因子分析（factor analysis）是另一种降维方法。与PCA不同的是，因子分析有假设而PCA没有假设。因子分析的基本假设是有一些隐藏特征与数据集的特征相关。

这个主题将浓缩（boil down）样本数据集的显性特征，尝试像理解因变量一样地理解自变量之间的隐藏特征。

Getting ready

让我们再用 `iris` 数据集来比较PCA与因子分析，首先加载因子分析类：

```
from sklearn import datasets
iris = datasets.load_iris()
from sklearn.decomposition import FactorAnalysis
```

How to do it...

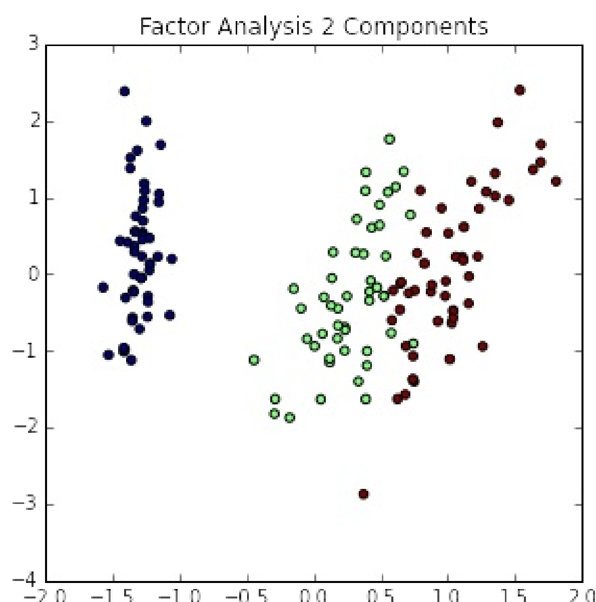
从编程角度看，两种方法没啥区别：

```
fa = FactorAnalysis(n_components=2)
iris_two_dim = fa.fit_transform(iris.data)
iris_two_dim[:5]
```

```
array([[-1.33125848,  0.55846779],
       [-1.33914102, -0.00509715],
       [-1.40258715, -0.307983  ],
       [-1.29839497, -0.71854288],
       [-1.33587575,  0.36533259]])
```

```
%matplotlib inline
from matplotlib import pyplot as plt
f = plt.figure(figsize=(5, 5))
ax = f.add_subplot(111)
ax.scatter(iris_two_dim[:,0], iris_two_dim[:, 1], c=iris.target)
ax.set_title("Factor Analysis 2 Components")
```

```
<matplotlib.text.Text at 0x8875ba8>
```



由于因子分析是一种概率性的转换方法，我们可以通过不同的角度来观察，例如模型观测值的对数似然估计值，通过模型比较对数似然估计值会更好。

因子分析也有不足之处。由于你不是通过拟合模型直接预测结果，拟合模型只是一个中间步骤。这本身并非坏事，但是训练实际模型时误差就会产生。

How it works...

因子分析与前面介绍的PCA类似。但两者有一个不同之处。PCA是通过对数据进行线性变换获取一个能够解释数据变量的主成分向量空间，这个空间中的每个主成分向量都是正交的。你可以把PCA看成是N维数据集降维成M维，其中$M \lt N$。

而因子分析的基本假设是，有M个重要特征和它们的线性组合（加噪声），能够构成原始的N维数据集。也就是说，你不需要指定结果变量（就是最终生成N维），而是要指定数据模型的因子数量（M个因子）。

1.11 用核PCA实现非线性降维

由于大多数统计方法最开始都是线性的，所以，想解决非线性问题，就需要做一些调整。PCA也是一种线性变换。本主题将首先介绍它的非线性形式，然后介绍如何降维。

Getting ready

如果数据都是线性的，生活得多容易啊，可惜现实并非如此。核主成分分析（Kernel PCA）可以处理非线性问题。数据先通过核函数（kernel function）转换成一个新空间，然后再用PCA处理。

要理解核函数之前，建议先尝试如何生成一个能够通过核PCA里的核函数线性分割的数据集。下面我们用余弦核（cosine kernel）演示。这个主题比前面的主题多一些理论。

How to do it...

余弦核可以用来比例样本空间中两个样本向量的夹角。当向量的大小（magnitude）用传统的距离度量不合适的时候，余弦核就有用了。

向量夹角的余弦公式如下：

$$cos(\theta)=\frac {A \cdot B} {{\begin{Vmatrix} A \end{Vmatrix}} {\begin{Vmatrix} B \end{Vmatrix}}}$$

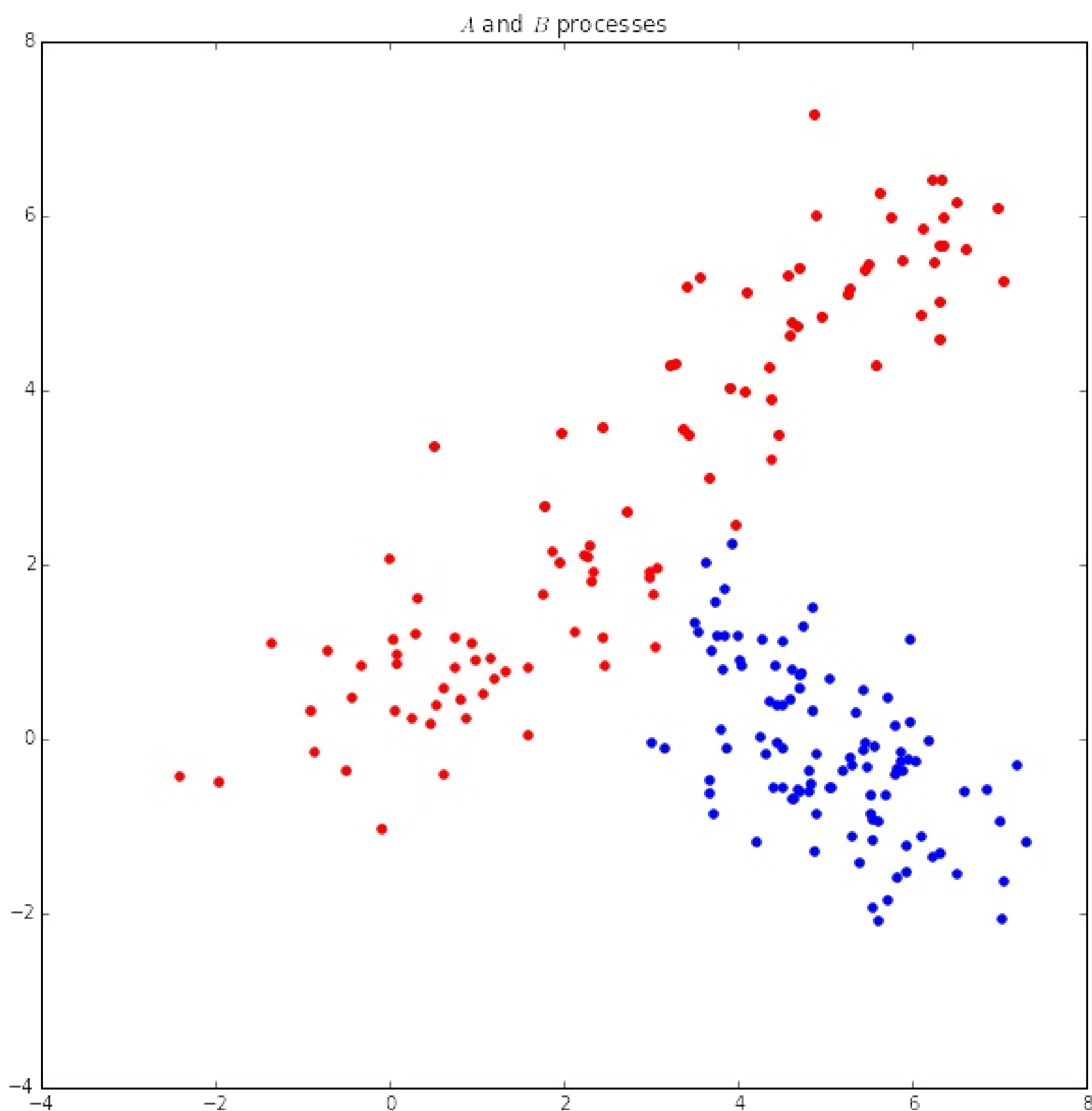
向量A和B夹角的余弦是两向量点积除以两个向量各自的L2范数。向量A和B的大小不会影响余弦值。

让我们生成一些数据来演示一下用法。首先，我们假设有两个不同的过程数据（process），称为A和B：

```
import numpy as np
A1_mean = [1, 1]
A1_cov = [[2, .99], [1, 1]]
A1 = np.random.multivariate_normal(A1_mean, A1_cov, 50)
A2_mean = [5, 5]
A2_cov = [[2, .99], [1, 1]]
A2 = np.random.multivariate_normal(A2_mean, A2_cov, 50)
A = np.vstack((A1, A2))
B_mean = [5, 0]
B_cov = [[.5, -1], [.9, -.5]]
B = np.random.multivariate_normal(B_mean, B_cov, 100)
```

```
import matplotlib.pyplot as plt
%matplotlib inline
f = plt.figure(figsize=(10, 10))
ax = f.add_subplot(111)
ax.set_title("$A$ and $B$ processes")
ax.scatter(A[:, 0], A[:, 1], color='r')
ax.scatter(A2[:, 0], A2[:, 1], color='r')
ax.scatter(B[:, 0], B[:, 1], color='b')
```

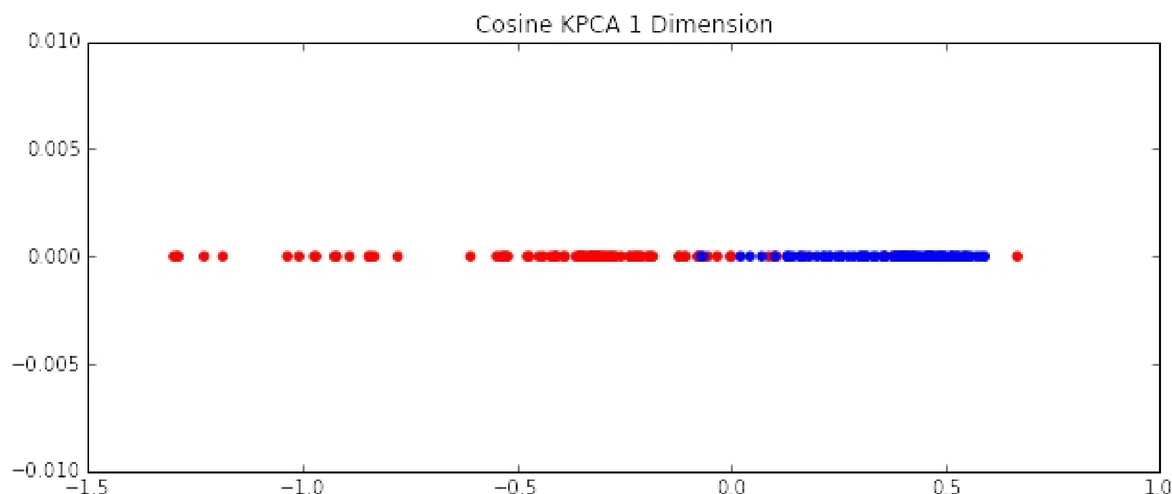
```
<matplotlib.collections.PathCollection at 0x73cd128>
```



上图看起来明显是两个不同的过程数据，但是用一超平面分割它们很难。因此，我们用前面介绍带余弦核的核PCA来处理：

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(kernel='cosine', n_components=1)
AB = np.vstack((A, B))
AB_transformed = kpca.fit_transform(AB)
```

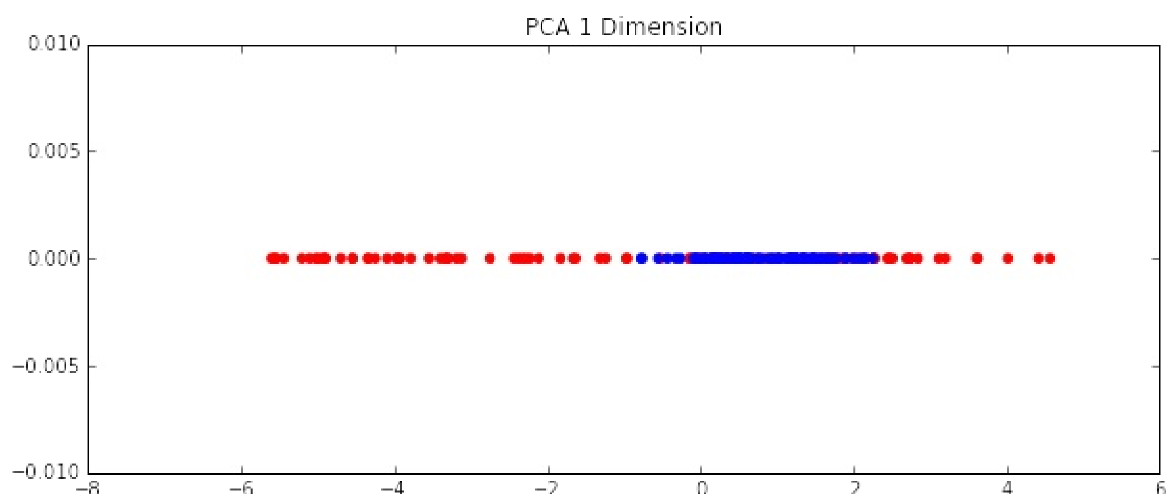
```
A_color = np.array(['r']*len(B))
B_color = np.array(['b']*len(B))
colors = np.hstack((A_color, B_color))
f = plt.figure(figsize=(10, 4))
ax = f.add_subplot(111)
ax.set_title("Cosine KPCA 1 Dimension")
ax.scatter(AB_transformed, np.zeros_like(AB_transformed), color=
colors);
```



用带余弦核的核PCA处理后，数据集变成了一维。如果用PCA处理就是这样：

```
from sklearn.decomposition import PCA
pca = PCA(1)
AB_transformed_Reg = pca.fit_transform(AB)
f = plt.figure(figsize=(10, 4))
ax = f.add_subplot(111)
ax.set_title("PCA 1 Dimension")
ax.scatter(AB_transformed_Reg, np.zeros_like(AB_transformed_Reg)
, color=colors)
```

```
<matplotlib.collections.PathCollection at 0x7c764a8>
```



很明显，核PCA降维效果更好。

How it works...

scikit-learn提供了几种像余弦核那样的核函数，也可以写自己的核函数。默认的函数有：

- 线性函数（linear）（默认值）
- 多项式函数（poly）
- 径向基函数（rbf，radial basis function）
- S形函数（sigmoid）
- 余弦函数（cosine）
- 用户自定义函数（precomputed）

还有一些因素会影响核函数的选择。例如，`degree` 参数可以设置 `poly` ，`rbf` 和 `sigmoid` 核函数的角度；而 `gamma` 会影响 `rbf` 和 `poly` 核，更多详情请查看 `KernelPCA` 文档。

后面关于支持向量机（SVM）的主题中将会进一步介绍 `rbf` 核函数。

需要注意的是：核函数处理非线性分离效果很好，但是一不小心就可能导致拟合过度。

1.12 用截断奇异值分解降维

截断奇异值分解（Truncated singular value decomposition，TSVD）是一种矩阵因式分解（factorization）技术，将矩阵M分解成U，Σ和V。它与PCA很像，只是SVD分解是在数据矩阵上进行，而PCA是在数据的协方差矩阵上进行。通常，SVD用于发现矩阵的主成份。

Getting ready

TSVD与一般SVD不同的是它可以产生一个指定维度的分解矩阵。例如，有一个$n \times n$矩阵，通过SVD分解后仍然是一个$n \times n$矩阵，而TSVD可以生成指定维度的矩阵。这样就可以实现降维了。

这里我们还用 `iris` 数据集来演示TSVD：

```
from sklearn.datasets import load_iris
iris = load_iris()
iris_data = iris.data
```

How to do it...

TSVD对象的用法和其他对象类似。首先导入需要的类，初始化，然后拟合：

```
from sklearn.decomposition import TruncatedSVD
```

```
svd = TruncatedSVD(2)
iris_transformed = svd.fit_transform(iris_data)
iris_data[:5]
```

```
array([[ 5.1,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  3.1,  1.5,  0.2],
       [ 5. ,  3.6,  1.4,  0.2]])
```

```
iris_transformed[:5]
```

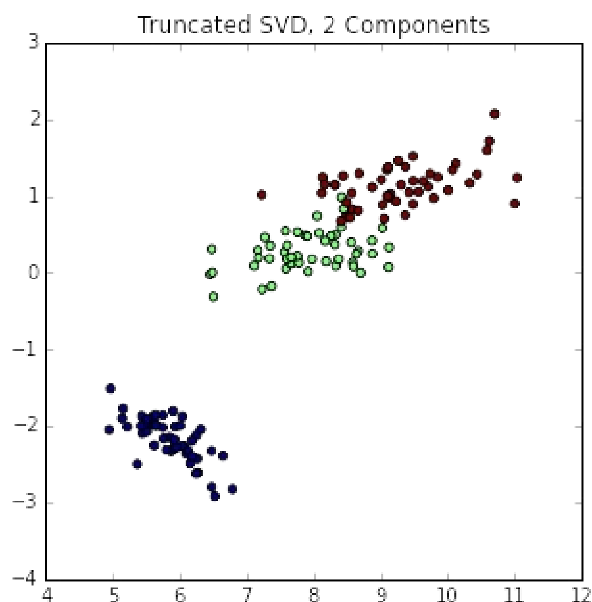
```
array([[ 5.91220352, -2.30344211],
       [ 5.57207573, -1.97383104],
       [ 5.4464847 , -2.09653267],
       [ 5.43601924, -1.87168085],
       [ 5.87506555, -2.32934799]])
```

最终结果如下图所示：

```
%matplotlib inline
import matplotlib.pyplot as plt
f = plt.figure(figsize=(5, 5))
ax = f.add_subplot(111)

ax.scatter(iris_transformed[:, 0], iris_transformed[:, 1], c=iri
s.target)
ax.set_title("Truncated SVD, 2 Components")
```

```
<matplotlib.text.Text at 0x8600be0>
```



How it works...

现在我们演示了scikit-learn的 `TruncatedSVD` 模块，让我们看看只用 `scipy` 学习一些细节。

首先，我们用 `scipy` 的 `linalg` 处理SVD：

```
import numpy as np
from scipy.linalg import svd
D = np.array([[1, 2], [1, 3], [1, 4]])
D
```

```
array([[1, 2],
       [1, 3],
       [1, 4]])
```

```
U, S, V = svd(D, full_matrices=False)
U.shape, S.shape, V.shape
```

```
((3, 2), (2,), (2, 2))
```

我们可以根据SVD的定义，用U，S和V还原矩阵D：

```
np.diag(S)
```

```
array([[ 5.64015854,  0.        ],
       [ 0.        ,  0.43429448]])
```

```
np.dot(U.dot(np.diag(S)), V)
```

```
array([[ 1.,  2.],
       [ 1.,  3.],
       [ 1.,  4.]])
```

`TruncatedSVD` 返回的矩阵是U和S的点积。如果我们想模拟TSVD，我们就去掉最新奇异值和对于U的列向量。例如，我们想要一个主成份，可以这样：

```
new_S = S[0]
new_U = U[:, 0]
new_U.dot(new_S)
```

```
array([-2.20719466, -3.16170819, -4.11622173])
```

一般情况下，如果我们想要截断维度t，那么我们就去掉$N-t$个奇异值。

There's more...

`TruncatedSVD` 还有一些细节需要注意。

符号翻转（Sign flipping）

`TruncatedSVD` 有个“陷阱”。随着随机数生成器状态的变化， `TruncatedSVD` 连续地拟合会改变输出的符合。为了避免这个问题，建议只用 `TruncatedSVD` 拟合一次，然后用其他变换。这正是管线命令的另一个用处。

要避免这种情况，可以这样：

```
tsvd = TruncatedSVD(2)
tsvd.fit(iris_data)
tsvd.transform(iris_data)[:5]
```

```
array([[ 5.91220352, -2.30344211],
       [ 5.57207573, -1.97383104],
       [ 5.4464847 , -2.09653267],
       [ 5.43601924, -1.87168085],
       [ 5.87506555, -2.32934799]])
```

稀疏矩阵

`TruncatedSVD` 相比PDA的一个优势是 `TruncatedSVD` 可以操作PDA处理不了的矩阵。这是因为PCA必须计算协方差矩阵，需要在整个矩阵上操作，如果矩阵太大，计算资源可能会不够用。

1.13 用字典学习分解法分类

在这个主题中，我们将介绍一种可以用于分类的分解方法——字典学习（Dictionary Learning），将数据集转换成一个稀疏的形式。

Getting ready

`DictionaryLearning` 方法的思想是把特征看作构成数据集的基础。首先我们导入 `iris` 数据集：

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
```

```
from sklearn.datasets import load_iris
iris = load_iris()
iris_data = iris.data
iris_target = iris.target
```

How to do it...

首先，导入 `DictionaryLearning` ：

```
from sklearn.decomposition import DictionaryLearning
```

然后用三个成分表示 `iris` 数据集中花的类型：

```
dl = DictionaryLearning(3)
```

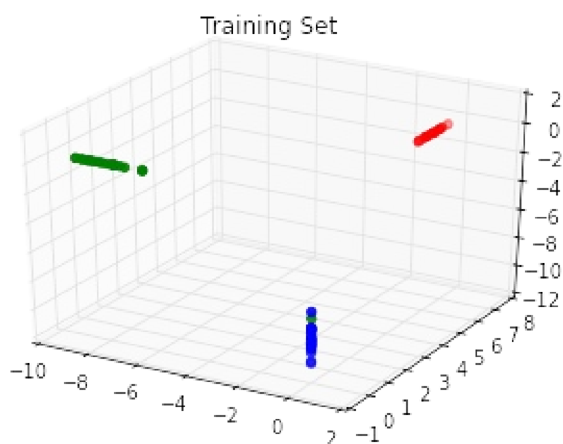
再用 `fit_transform` 转换其他数据，这样我们就可以对比训练前后的数据了：

```
transformed = dl.fit_transform(iris_data[::2])
transformed[:5]
```

```
array([[ 0.        ,  6.34476574,  0.        ],
       [ 0.        ,  5.83576461,  0.        ],
       [ 0.        ,  6.32038375,  0.        ],
       [ 0.        ,  5.89318572,  0.        ],
       [ 0.        ,  5.45222715,  0.        ]])
```

我们可以可视化这个结果。注意，每个成分的值分别平行x，y和z三个轴，其他坐标都是0；这就是稀疏性。

```
from mpl_toolkits.mplot3d import Axes3D
colors = np.array(list('rgb'))
f = plt.figure()
ax = f.add_subplot(111, projection='3d')
ax.set_title("Training Set")
ax.scatter(transformed[:, 0], transformed[:, 1], transformed[:, 2
], color=colors[iris.target[::2]]);
```

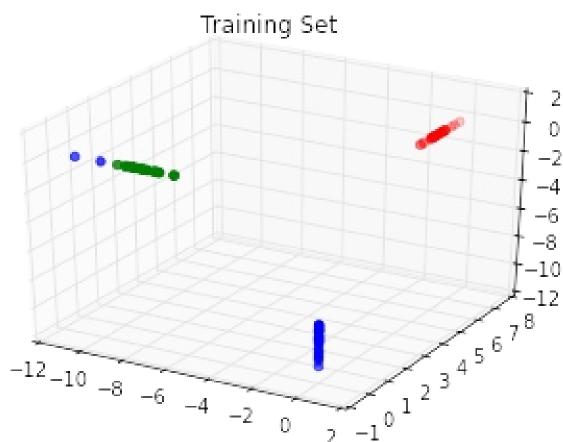


如果你细心看，还是会发现一些误差。有一个样本分错了类型，虽然一个错误并不是很严重。

下面，让我们用 `fit` 而不用 `fit_transform` 来训练数据集：

```
transformed = dl.transform(iris_data[1::2])
```

```
colors = np.array(list('rgb'))
f = plt.figure()
ax = f.add_subplot(111, projection='3d')
ax.set_title("Training Set")
ax.scatter(transformed[:, 0], transformed[:, 1], transformed[:, 2
], color=colors[iris.target[1::2]]);
```



还是有一些分类错误的样本。如果你看看之前降维主题中的图，会发现绿色和蓝色两类数据有交叉部分。

How it works...

`DictionaryLearning`具有信号处理和神经学领域的背景知识。其理念是某一时刻只有少数特征可以实现。因此，`DictionaryLearning`在假设大多数特征都是0的情况下，尝试发现一个适当的数据表现形式。

1.14 用管线命令连接多个转换方法

下面，让我们用管线命令连接多个转换方法，来演示一个复杂点儿的例子。

Getting ready

本主题将再度释放管线命令的光芒。之前我们用它处理缺失数据，只是牛刀小试罢了。下面我们用管线命令把多个预处理步骤连接起来处理，会非常方便。

首先，我们加载带缺失值的`iris`数据集：

```
from sklearn.datasets import load_iris
import numpy as np

iris = load_iris()
iris_data = iris.data

mask = np.random.binomial(1, .25, iris_data.shape).astype(bool)
iris_data[mask] = np.nan

iris_data[:5]
```

```
array([[ nan,  3.5,  1.4,  0.2],
       [ 4.9,  3. ,  1.4,  0.2],
       [ 4.7,  3.2,  1.3,  0.2],
       [ 4.6,  nan,  1.5,  nan],
       [ 5. ,  3.6,  1.4,  0.2]])
```

How to do it...

本主题的目标是首先补全`iris_data`的缺失值，然后对补全的数据集用PCA。可以看出这个流程需要一个训练数据集和一个对照集（holdout set）；管线命令会让事情更简单，不过之前我们做一些准备工作。

首先加载需要的模块：

```
from sklearn import pipeline, preprocessing, decomposition
```

然后，建立 `Imputer` 和 `PCA` 类：

```
pca = decomposition.PCA()
imputer = preprocessing.Imputer()
```

有了两个类之后，我们就可以用管线命令处理：

```
pipe = pipeline.Pipeline([('imputer', imputer), ('pca', pca)])
np.set_printoptions(2)
iris_data_transformed = pipe.fit_transform(iris_data)
iris_data_transformed[:5]
```

```
array([[-2.44, -0.79, -0.12, -0.1 ],
       [-2.67,  0.2 , -0.21,  0.15],
       [-2.83,  0.31, -0.19, -0.08],
       [-2.35,  0.66,  0.67, -0.06],
       [-2.68, -0.06, -0.2 , -0.4 ]])
```

如果我们用单独的步骤分别处理，每个步骤都要用一次 `fit_transform`，而这里只需要用一次，而且只需要一个对象。

How it works...

管线命令的每个步骤都是用一个元组表示，元组的第一个元素是对象的名称，第二个元素是对象。

本质上，这些步骤都是在管线命令调用时依次执行 `fit_transform` 方法。还有一种快速但不太简洁的管线命令建立方法，就像我们快速建立标准化调整模型一样，只不过用 `StandardScaler` 会获得更多功能。`pipeline` 函数将自动创建管线命令的名称：

```
pipe2 = pipeline.make_pipeline(imputer, pca)
pipe2.steps
```

```
[('imputer',
  Imputer(axis=0, copy=True, missing_values='NaN', strategy='mea
n', verbose=0)),
 ('pca', PCA(copy=True, n_components=None, whiten=False))]
```

这和前面的模型结果一样：

```
iris_data_transformed2 = pipe2.fit_transform(iris_data)
iris_data_transformed2[:5]
```

```
array([[-2.44, -0.79, -0.12, -0.1 ],
       [-2.67,  0.2 , -0.21,  0.15],
       [-2.83,  0.31, -0.19, -0.08],
       [-2.35,  0.66,  0.67, -0.06],
       [-2.68, -0.06, -0.2 , -0.4 ]])
```

There's more...

管线命令连接内部每个对象的属性是通过`set_params`方法实现，其参数用`<对象名称>__<对象参数>`表示。例如，我们设置PCA的主成份数量为2：

```
pipe2.set_params(pca__n_components=2)
```

```
Pipeline(steps=[('imputer', Imputer(axis=0, copy=True, missing_v
alues='NaN', strategy='mean', verbose=0)), ('pca', PCA(copy=True
, n_components=2, whiten=False))])
```

> `__`标识在Python社区读作**dunder**。

这里`n_components=2`是`pca`本身的参数。我们再演示一下，输出将是一个$N \times 2$维矩阵：

```
iris_data_transformed3 = pipe2.fit_transform(iris_data)
iris_data_transformed3[:5]
```

```
array([[-2.44, -0.79],
       [-2.67,  0.2 ],
       [-2.83,  0.31],
       [-2.35,  0.66],
       [-2.68, -0.06]])
```

1.15 用正态随机过程处理回归

这个主题将介绍如何用正态随机过程（Gaussian process，GP）处理回归问题。在线性模型部分，我们曾经见过在变量间可能存在相关性时，如何用贝叶斯岭回归(Bayesian Ridge Regression)表示先验概率分布（prior）信息。

正态分布过程关心的是方程而不是均值。但是，如果我们假设一个正态分布的均值为0，那么我们需要确定协方差。

这样处理就与线性回归问题中先验概率分布可以用相关系数表示的情况类似。用GP处理的先验就可以用数据、样本数据间协方差构成函数表示，因此必须从数据中拟合得出。具体内容参考The Gaussian Processes Web Site。

Getting ready

首先要我们用一些数据来演示用scikit-learn处理GP：

```
import numpy as np
from sklearn.datasets import load_boston
boston = load_boston()
boston_X = boston.data
boston_y = boston.target
train_set = np.random.choice([True, False], len(boston_y), p=[.75
, .25])
```

How to do it...

有了数据之后，我们就创建scikit-learn的 `GaussianProcess` 对象。默认情况下，它使用一个常系数回归方程（constant regression function）和平方指数相关函数（ squared exponential correlation），是最主流的选择之一：

```
from sklearn.gaussian_process import GaussianProcess
gp = GaussianProcess()
gp.fit(boston_X[train_set], boston_y[train_set])
```

```
GaussianProcess(beta0=None,
        corr=<function squared_exponential at 0x0000000006F1C950
>,
        normalize=True, nugget=array(2.220446049250313e-15),
        optimizer='fmin_cobyla', random_start=1,
        random_state=<mtrand.RandomState object at 0x00000000052
A4BA8>,
        regr=<function constant at 0x0000000006F15950>,
        storage_mode='full', theta0=array([[ 0.1]]), thetaL=None
,
        thetaU=None, verbose=False)
```

其中，

- `beta0` ：回归权重。默认是用MLE（最大似然估计，Maximum Likelihood

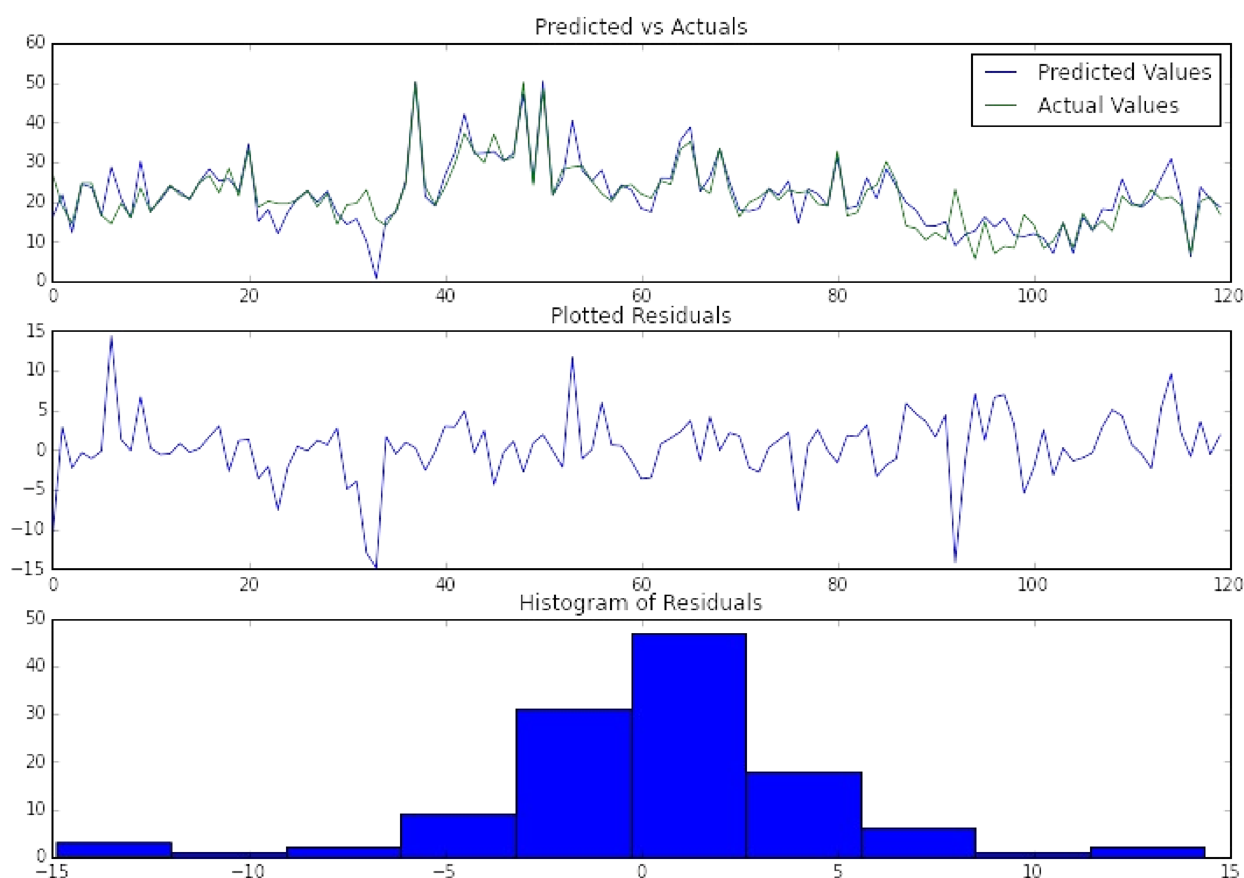
Estimation）方法进行估计。
- `corr` ：相关系数方程。提供了若干种方程，后面会介绍。
- `normalize` ：默认是 `True` ，中性化调整样本值，方便应用MLE进行估计。
- `nugget` ：正则化参数，是可选的，默认是一个很小的值。你可以将这个参数用于每个样本值（参数是一个数值），也可以对样本值使用不同的参数（参数是一个数组，与样本值个数相等）。
- `regr` ：默认是常系数回归方程。

现在让我们拟合对象看看测试效果：

```
test_preds = gp.predict(boston_X[~train_set])
```

让我们把预测值和实际值画出来比较一下。因为我们做了回归，还可以看看残差散点图和残差直方图。

```
%matplotlib inline
from matplotlib import pyplot as plt
f, ax = plt.subplots(figsize=(10, 7), nrows=3)
f.tight_layout()
ax[0].plot(range(len(test_preds)), test_preds, label='Predicted
Values');
ax[0].plot(range(len(test_preds)), boston_y[~train_set], label='
Actual Values');
ax[0].set_title("Predicted vs Actuals")
ax[0].legend(loc='best')
ax[1].plot(range(len(test_preds)),
test_preds - boston_y[~train_set]);
ax[1].set_title("Plotted Residuals")
ax[2].hist(test_preds - boston_y[~train_set]);
ax[2].set_title("Histogram of Residuals");
```



How it works...

上面我们快速演示了一下，现在让我们看看这些参数，看看如何优化它们。首先，我们看看 `corr` 函数的类型。这个函数描述了不同组 `X` 之间的相关性。scikit-learn提供了5种函数类型：

- 绝对值指数函数（absolute_exponential）
- 平方指数函数（squared_exponential）
- 广义指数函数（generalized_exponential）
- 立方项函数（cubic）
- 线性函数（linear）

例如，平方指数函数公式如下：

$$K= \exp {-\frac {|d|^2}{2l^2}}$$

另外，线性函数就是两个点的点积：

$$K=x^T{x'}$$

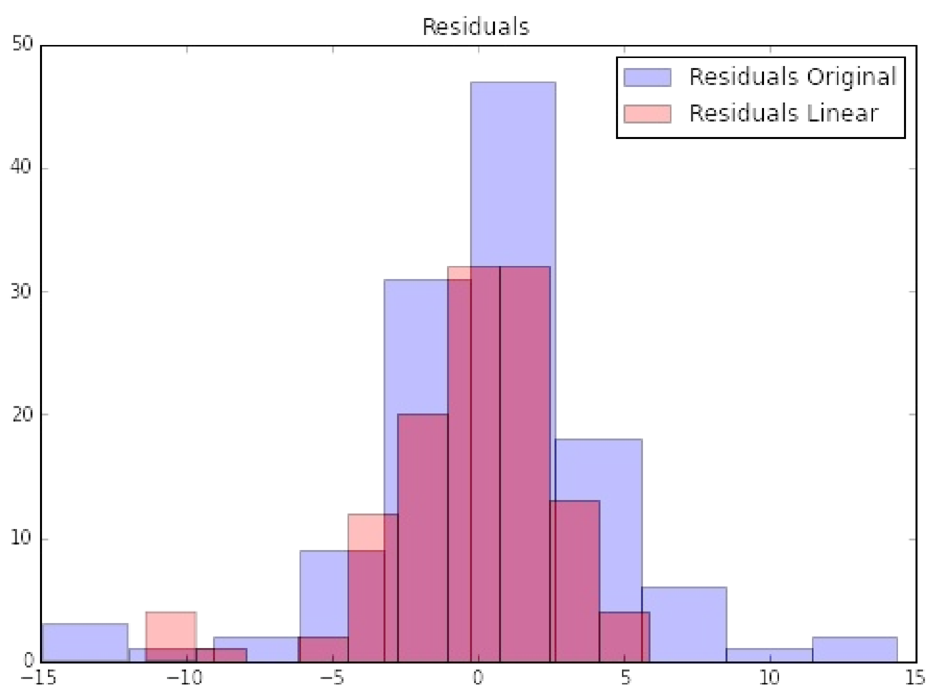
另一个参数是 `theta0` ，表示参数估计的起始点。

一旦我们有了K和均值的估计值，过程就完全确定了，因为它是一个GP；之所以用正态分布，是因为在机器学习中它一直很受欢迎。

下面我们换个 `regr` 函数类型和 `theta0` 参数，看看结果会如何变化：

```
gp = GaussianProcess(regr='linear', theta0=5e-1)
gp.fit(boston_X[train_set], boston_y[train_set]);
linear_preds = gp.predict(boston_X[~train_set])
```

```
f, ax = plt.subplots(figsize=(7, 5))
f.tight_layout()
ax.hist(test_preds - boston_y[~train_set],label='Residuals Origi
nal', color='b', alpha=.5);
ax.hist(linear_preds - boston_y[~train_set],label='Residuals Lin
ear', color='r', alpha=.5);
ax.set_title("Residuals")
ax.legend(loc='best');
```



很明显，第二个模型的预测效果大部分区间要更好。如果我们把残差汇总起来，我们可以看看MSE预测的结果：

```
np.power(test_preds - boston_y[~train_set], 2).mean()
```

```
17.456331927446904
```

```
np.power(linear_preds - boston_y[~train_set], 2).mean()
```

```
9.320038747573518
```

There's more...

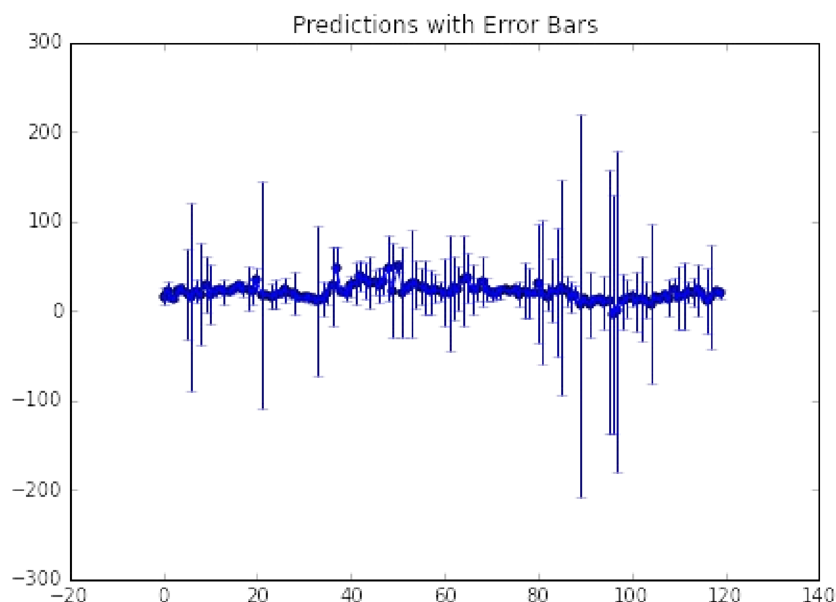
我们可能还想掌握估计的不确定性。在我们预测的时候，如果我们 `eval_MSE` 设置为 `True` ，我们就获得MSE的值，这时预测返回的是预测值与MSE估计值的元组。

```
test_preds, MSE = gp.predict(boston_X[~train_set], eval_MSE=True
)
MSE[:5]
```

```
array([ 5.14026315,  5.30852052,  0.91152632,  2.87148688,  2.55
714482])
```

这样我们就可以计算估计的误差了。让我们画出来看看准确率：

```
f, ax = plt.subplots(figsize=(7, 5))
n = 120
rng = range(n)
ax.scatter(rng, test_preds[:n])
ax.errorbar(rng, test_preds[:n], yerr=1.96*MSE[:n])
ax.set_title("Predictions with Error Bars")
ax.set_xlim((-20, 140));
```



你会看到，许多点的估计都有些变化。但是，前面的数据显示，总体误差不是特别大。

1.16 直接定义一个正态随机过程对象

前面我们只触及了正态随机过程的表面。在本主题中，我们将介绍直接创建一个具有指定相关函数的正态随机过程。

Getting ready

`gaussian_process` 模块可以直接连接不同的相关函数与回归方程。这样就可以不创建 `GaussianProcess` 对象，直接通过函数创建需要的对象。如果你更熟悉面向对象的编程方法，这里只算是模块级的一个类方法而已。

在本主题中，我们将使用大部分函数，并把他们的结果用几个例子显示出来。如果你想真正掌握这些相关函数的特点，不要仅仅停留在这些例子上。这里不再介绍新的数学理论，让我们直接演示如何做。

How to do it...

首先，我们导入要回归的数据：

```
from sklearn.datasets import make_regression
X, y = make_regression(1000, 1, 1)
from sklearn.gaussian_process import regression_models
```

第一个相关函数是常系数相关函数。它有若干常数构成：

```
regression_models.constant(X)[:5]
```

```
array([[ 1.],
       [ 1.],
       [ 1.],
       [ 1.],
       [ 1.]])
```

还有线性相关函数与平方指数相关函数，它们也是 `GaussianProcess` 类的默认值：

```
regression_models.linear(X)[:1]
```

```
array([[ 1.        , -1.29786999]])
```

```
regression_models.quadratic(X)[:1]
```

```
array([[ 1.        , -1.29786999,  1.68446652]])
```

How it works...

这样我们就可以得到回归函数了，可以直接用 `GaussianProcess` 对象来处理它们。默认值是常系数相关函数，但我们也可以把轻松的把线性模型和平方指数模型传递进去。

1.17 用随机梯度下降处理回归

本主题将介绍随机梯度下降法（Stochastic Gradient Descent，SGD），我们将用它解决回归问题，后面我们还用它处理分类问题。

Getting ready

SGD是机器学习中的无名英雄（unsung hero），许多算法的底层都有SGD的身影。之所以受欢迎是因为其简便与快速——处理大量数据时这些都是好事儿。

SGD成为许多机器学习算法的核心的另一个原因是它很容易描述过程。在本章的最后，我们对数据作一些变换，然后用模型的损失函数（loss function）拟合数据。

How to do it...

如果SGD适合处理大数据集，我们就用大点儿的数据集来演示：

```
from sklearn import datasets
X, y = datasets.make_regression(int(1e6))
print("{:,}".format(int(1e6)))
```

```
1,000,000
```

值得进一步了解数据对象的构成和规模信息。还在我们用的是NumPy数组，所以我们可以获得 `nbytes` 。Python本身没有获取NumPy数组大小的方法。输出结果与系统有关，你的结果和下面的数据可能不同：

```
print("{:,}".format(X.nbytes))
```

```
800,000,000
```

我们把字节码 `nbytes` 转换成MB（megabytes），看着更直观：

```
X.nbytes / 1e6
```

```
800.0
```

因此，每个数据点的字节数就是：

```
X.nbytes / (X.shape[0] * X.shape[1])
```

```
8.0
```

这些信息和我们的目标没多大关系，不过了解数据对象的构成和规模信息还是值得的。

现在，我们有了数据，就用 `SGDRegressor` 来拟合：

```
import numpy as np
from sklearn import linear_model
sgd = linear_model.SGDRegressor()
train = np.random.choice([True, False], size=len(y), p=[.75, .25
])
sgd.fit(X[train], y[train])
```

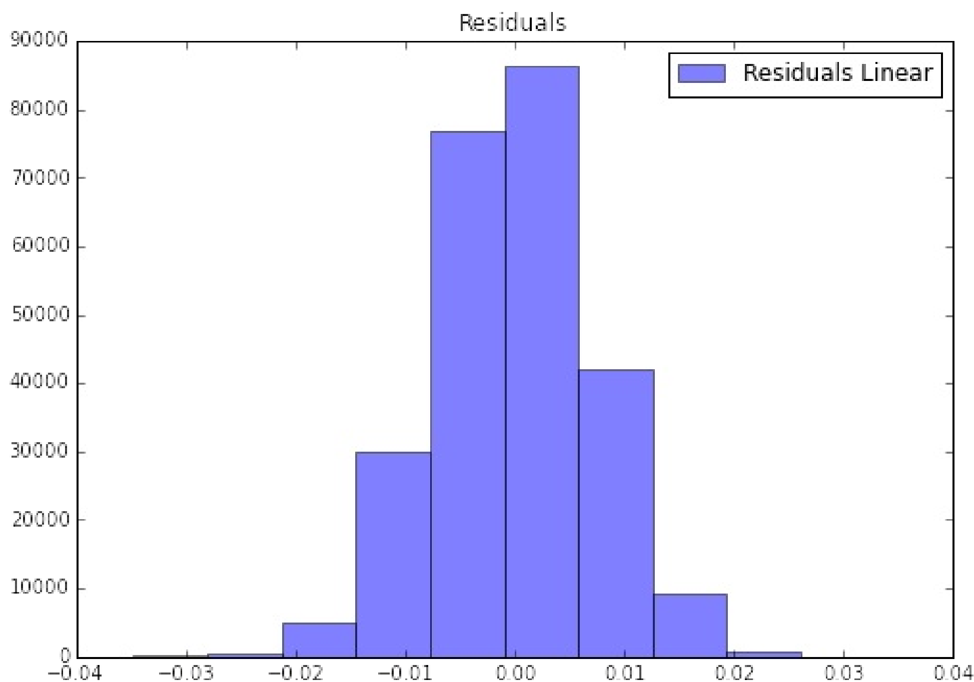
```
SGDRegressor(alpha=0.0001, average=False, epsilon=0.1, eta0=0.01
,
       fit_intercept=True, l1_ratio=0.15, learning_rate='invscal
ing',
       loss='squared_loss', n_iter=5, penalty='l2', power_t=0.25
,
       random_state=None, shuffle=True, verbose=0, warm_start=Fa
lse)
```

这里又出现一个“充实的（beefy）”对象。重点需要了解我们的损失函数是 `squared_loss`，与线性回归里的残差平方和是一样的。还需要注意 `shuffle` 会对数据产生随机搅动（shuffle），这在解决伪相关问题时很有用。

scikit-learn用 `fit_intercept` 方法可以自动加一列1。如果想从拟合结果中看到很多输出，就把 `verbose` 设为1。用scikit-learn的API预测，我们可以统计残差的分布情况：

```
linear_preds = sgd.predict(X[~train])
```

```
%matplotlib inline
from matplotlib import pyplot as plt
f, ax = plt.subplots(figsize=(7, 5))
f.tight_layout()
ax.hist(linear_preds - y[~train],label='Residuals Linear', color=
'b', alpha=.5);
ax.set_title("Residuals")
ax.legend(loc='best');
```



拟合的效果非常好。异常值很少，直方图也呈现出完美的正态分布钟形图。

How it works...

当然这里我们用的是虚拟数据集，但是你也可以用更大的数据集合。例如，如果你在华尔街工作，有可能每天一个市场都有20亿条交易数据。现在如果有一周或一年的数据，用SGD算法就可能无法运行了。很难处理这么大的数据量，因为标准的梯度下降法每一步都要计算梯度，计算量非常庞大。

标准的梯度下降法的思想是在每次迭代计算一个新的相关系数矩阵，然后用学习速率（learning rate）和目标函数（objective function）的梯度调整它，直到相关系数矩阵收敛为止。如果用伪代码写就是这样：

```
while not_converged:
    w = w – learning_rate * gradient(cost(w))
```

这里涉及的变量包括：

- `w` ：相关系数矩阵
- `learning_rate` ：每次迭代时前进的长度。如果收敛效果不好，调整这个参数很重要
- `gradient` ：导数矩阵
- `cost` ：回归的残差平方和。后面我们会介绍，不同的分类方法中损失函数定义不同，具有可变性也是SGD应用广泛的理由之一。

除了梯度函数有点复杂之外，这个方法还是可以的。随着相关系数向量的增加，梯度的计算也会变得越来越慢。每次更新之前，我们都需要对每个数据点计算新权重。

SGD的工作方式稍有不同；每次迭代不是批量更新梯度，而是只更新新数据点的参数。这些数据点是随机选择的，因此称为随机梯度下降法。

第二章 处理线性模型

> 作者：Trent Hauck
>
> 译者：muxuezi
>
>

本章包括以下主题：

简介

线性模型是统计学和机器学习的基础。很多方法都利用变量的线性组合描述数据之间的关系。通常都要花费很大精力做各种变换，目的就是为了让数据可以描述成一种线性组合形式。

本章，我们将从最简单的数据直线拟合模型到分类模型，最后介绍贝叶斯岭回归。

2.1 线性回归模型

现在，我们来做一些建模！我们从最简单的线性回归（Linear regression）开始。线性回归是最早的也是最基本的模型——把数据拟合成一条直线。

Getting ready

`boston` 数据集很适合用来演示线性回归。 `boston` 数据集包含了波士顿地区的房屋价格中位数。还有一些可能会影响房价的因素，比如犯罪率（crime rate）。

首先，让我们加载数据：

```
from sklearn import datasets
boston = datasets.load_boston()
```

How to do it...

实际上，用scikit-learn的线性回归非常简单，其API和前面介绍的模型一样。

首先，导入 `LinearRegression` 类创建一个对象：

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
```

现在，再把自变量和因变量传给 `LinearRegression` 的 `fit` 方法：

```
lr.fit(boston.data, boston.target)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, norm
alize=False)
```

2.2 评估线性回归模型

在这个主题中，我们将介绍回归模型拟合数据的效果。上一个主题我们拟合了数据，但是并没太关注拟合的效果。每当拟合工作做完之后，我们应该问的第一个问题就是“拟合的效果如何？”本主题将回答这个问题。

Getting ready

我们还用上一主题里的 `lr` 对象和 `boston` 数据集。 `lr` 对象已经拟合过数据，现在有许多方法可以用。

```
from sklearn import datasets
boston = datasets.load_boston()
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(boston.data, boston.target)
```

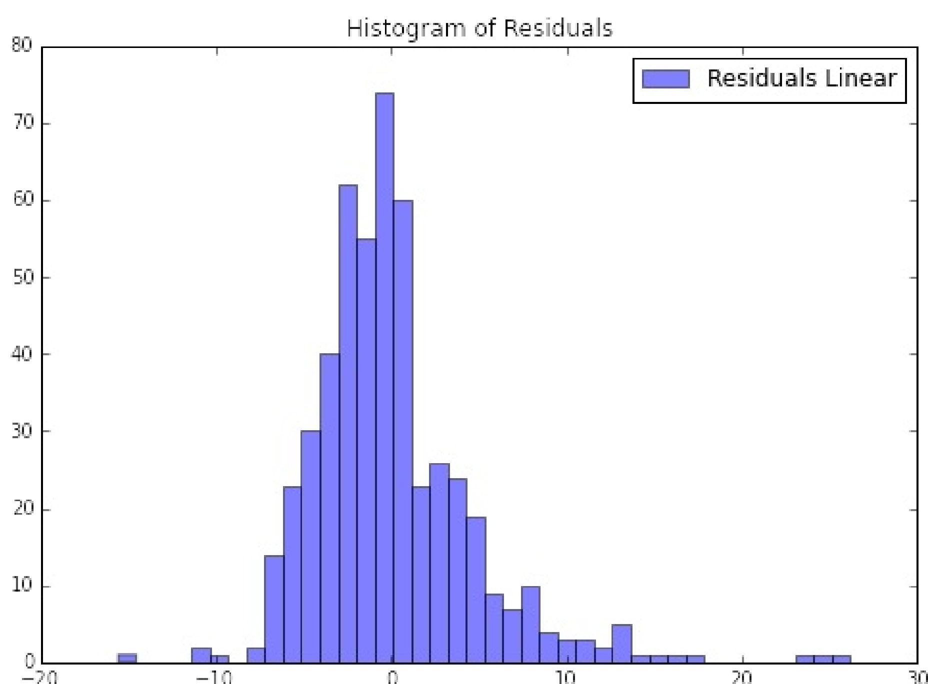
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, norm
alize=False)
```

```
predictions = lr.predict(boston.data)
```

How to do it...

我们可以看到一些简单的量度（metris）和图形。让我们看看上一章的残差图：

```
%matplotlib inline
from matplotlib import pyplot as plt
f, ax = plt.subplots(figsize=(7, 5))
f.tight_layout()
ax.hist(boston.target - predictions,bins=40, label='Residuals Li
near', color='b', alpha=.5);
ax.set_title("Histogram of Residuals")
ax.legend(loc='best');
```



如果你用IPython Notebook，就用 `%matplotlib inline` 命令在网页中显示matplotlib图形。如果你不用，就用 `f.savefig('myfig.png')` 保存图形，以备使用。

> 画图的库是matplotlib，并非本书重点，但是可视化效果非常好。

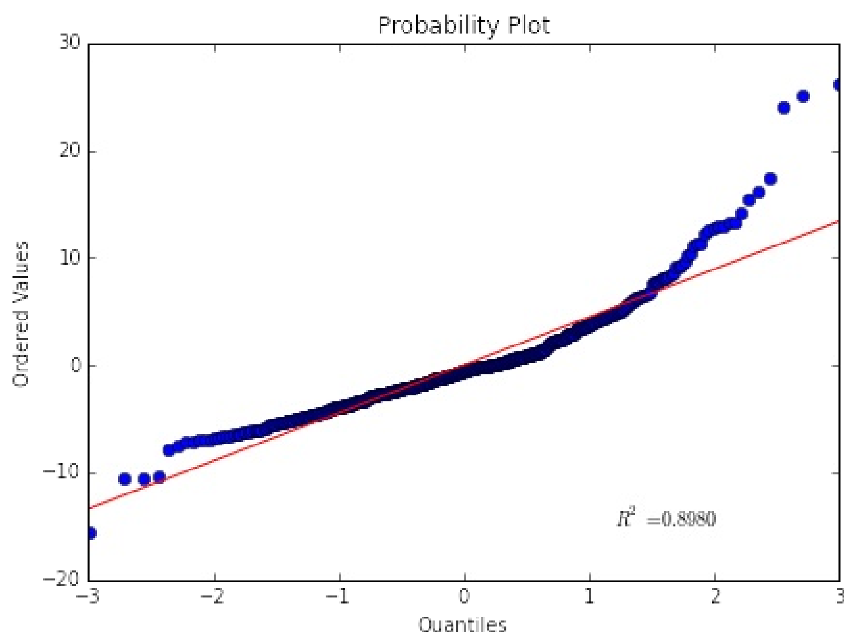
和之前介绍的一样，误差项服从均值为0的正态分布。残差就是误差，所以这个图也应噶近似正态分布。看起来拟合挺好的，只是有点偏。我们计算一下残差的均值，应该很接近0：

```
import numpy as np
np.mean(boston.target - predictions)
```

```
6.0382090193051989e-16
```

另一个值得看的图是**Q-Q**图（分位数概率分布），我们用Scipy来实现图形，因为它内置这个概率分布图的方法：

```
from scipy.stats import probplot
f = plt.figure(figsize=(7, 5))
ax = f.add_subplot(111)
probplot(boston.target - predictions, plot=ax);
```



这个图里面倾斜的数据比之前看的要更清楚一些。

我们还可以观察拟合其他量度，最常用的还有均方误差（mean squared error，MSE），平均绝对误差（mean absolute deviation，MAD）。让我们用Python实现这两个量度。后面我们用scikit-learn内置的量度来评估回归模型的效果：

```
def MSE(target, predictions):
    squared_deviation = np.power(target - predictions, 2)
    return np.mean(squared_deviation)
```

```
MSE(boston.target, predictions)
```

```
21.897779217687496
```

```
def MAD(target, predictions):
    absolute_deviation = np.abs(target - predictions)
    return np.mean(absolute_deviation)
```

```
MAD(boston.target, predictions)
```

```
3.2729446379969396
```

How it works...

MSE的计算公式是：

$$E(\hat y_t - y_i)^2$$

计算预测值与实际值的差，平方之后再求平均值。这其实就是我们寻找最佳相关系数时是目标。高斯—马尔可夫定理（Gauss-Markov theorem）实际上已经证明了线性回归的回归系数的最佳线性无偏估计（BLUE）就是最小均方误差的无偏估计（条件是误差变量不相关，0均值，同方差）。在用岭回归弥补线性回归的不足主题中，我们会看到，当我们的相关系数是有偏估计时会发生什么。

MAD是平均绝对误差，计算公式为：

$$E|\hat y_t - y_i|$$

线性回归的时候MAD通常不用，但是值得一看。为什么呢？可以看到每个量度的情况，还可以判断哪个量度更重要。例如，用MSE，较大的误差会获得更大的惩罚，因为平方把它放大。

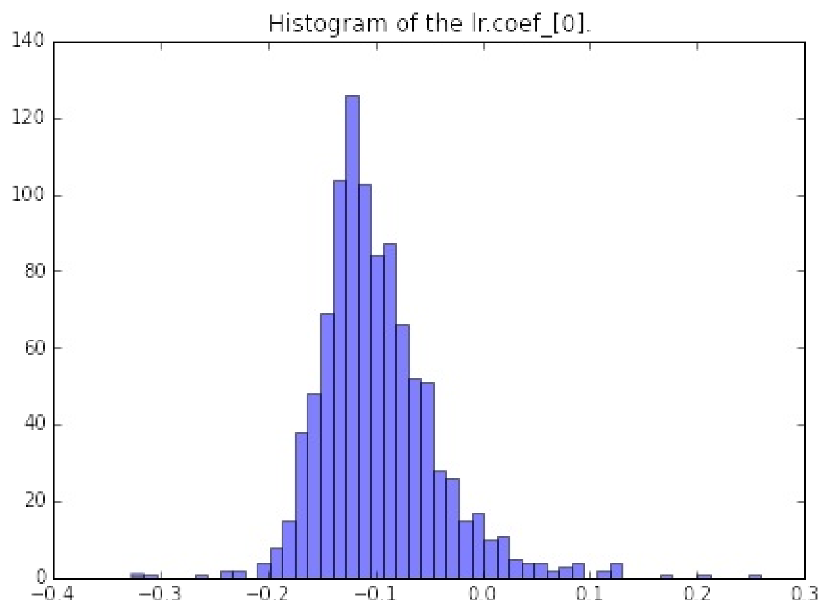
There's more...

还有一点需要说明，那就是相关系数是随机变量，因此它们是有分布的。让我们用bootstrapping（重复试验）来看看犯罪率的相关系数的分布情况。bootstrapping是一种学习参数估计不确定性的常用手段：

```
n_bootstraps = 1000
len_boston = len(boston.target)
subsample_size = np.int(0.5*len_boston)
subsample = lambda: np.random.choice(np.arange(0, len_boston),si
ze=subsample_size)
coefs = np.ones(n_bootstraps) #相关系数初始值设为1
for i in range(n_bootstraps):
    subsample_idx = subsample()
    subsample_X = boston.data[subsample_idx]
    subsample_y = boston.target[subsample_idx]
    lr.fit(subsample_X, subsample_y)
    coefs[i] = lr.coef_[0]
```

我们可以看到这个相关系数的分布直方图：

```
f = plt.figure(figsize=(7, 5))
ax = f.add_subplot(111)
ax.hist(coefs, bins=50, color='b', alpha=.5)
ax.set_title("Histogram of the lr.coef_[0].");
```



我们还想看看重复试验后的置信区间：

```
np.percentile(coefs, [2.5, 97.5])
```

```
array([-0.18030624,  0.03816062])
```

置信区间的范围表面犯罪率其实不影响房价，因为0在置信区间里面，表面犯罪率可能与房价无关。

值得一提的是，bootstrapping可以获得更好的相关系数估计值，因为使用bootstrapping方法的均值，会比普通估计方法更快地收敛（**converge**）到真实均值。

2.3 用岭回归弥补线性回归的不足

本主题将介绍岭回归。和线性回归不同，它引入了正则化参数来“缩减”相关系数。当数据集中存在共线因素时，岭回归会很有用。

Getting ready

让我们加载一个不满秩（low effective rank）数据集来比较岭回归和线性回归。秩是矩阵线性无关组的数量，满秩是指一个$m \times n$矩阵中行向量或列向量中现行无关组的数量等于$min(m,n)$。

How to do it...

首先我们用`make_regression`建一个有3个自变量的数据集，但是其秩为2，因此3个自变量中有两个自变量存在相关性。

```
from sklearn.datasets import make_regression
reg_data, reg_target = make_regression(n_samples=2000, n_feature
s=3, effective_rank=2, noise=10)
```

首先，我们用普通的线性回归拟合：

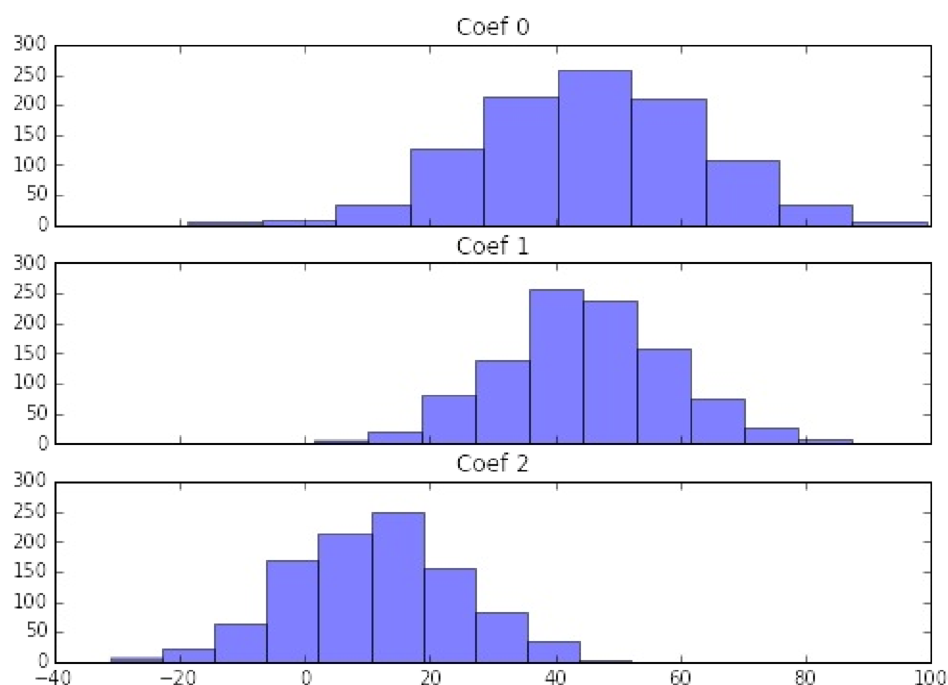
```
import numpy as np
from sklearn.linear_model import LinearRegression
lr = LinearRegression()

def fit_2_regression(lr):
    n_bootstraps = 1000
    coefs = np.ones((n_bootstraps, 3))
    len_data = len(reg_data)
    subsample_size = np.int(0.75*len_data)
    subsample = lambda: np.random.choice(np.arange(0, len_data),
 size=subsample_size)

    for i in range(n_bootstraps):
        subsample_idx = subsample()
        subsample_X = reg_data[subsample_idx]
        subsample_y = reg_target[subsample_idx]
        lr.fit(subsample_X, subsample_y)
        coefs[i][0] = lr.coef_[0]
        coefs[i][1] = lr.coef_[1]
        coefs[i][2] = lr.coef_[2]
    %matplotlib inline
    import matplotlib.pyplot as plt
    f, axes = plt.subplots(nrows=3, sharey=True, sharex=True, fi
gsize=(7, 5))
    f.tight_layout()

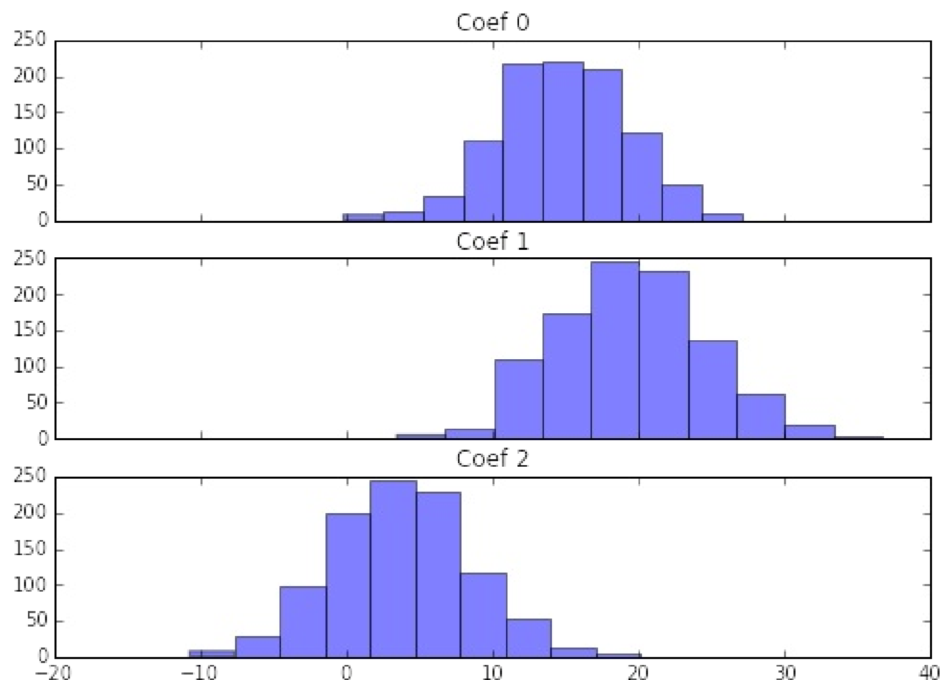
    for i, ax in enumerate(axes):
        ax.hist(coefs[:, i], color='b', alpha=.5)
        ax.set_title("Coef {}".format(i))
    return coefs

coefs = fit_2_regression(lr)
```



我们再用 `Ridge` 来拟合数据，对比结果：

```
from sklearn.linear_model import Ridge
coefs_r = fit_2_regression(Ridge())
```



两个回归算法的结果看着好像差不多，其实不然。岭回归的相关系数更接近0。让我们看看两者相关系数的差异：

```
np.mean(coefs - coefs_r, axis=0)
```

```
array([ 30.54319761,  25.1726559 ,   7.40345307])
```

从均值上看，线性回归比岭回归的相关系数要更很多。均值显示的差异其实是线性回归的相关系数隐含的偏差。那么，岭回归究竟有什么好处呢？让我们再看看相关系数的方差：

```
np.var(coefs, axis=0)
```

```
array([ 302.16242654,  177.36842779,  179.33610289])
```

```
np.var(coefs_r, axis=0)
```

```
array([ 19.60727206,  25.4807605 ,  22.74202917])
```

岭回归的相关系数方差也会小很多。这就是机器学习里著名的偏差-方差均衡(Bias-Variance Trade-off)。下一个主题我们将介绍如何调整岭回归的参数正则化，那是偏差-方差均衡的核心内容。

How it works...

介绍参数正则化之前，我们总结一下岭回归与线性回归的不同。前面介绍过，线性回归的目标是最小化 $ {\begin{Vmatrix} \hat y - X \beta \end{Vmatrix}}^2 $。

岭回归的目标是最小化 $ {\begin{Vmatrix} \hat y - X \beta \end{Vmatrix}}^2 + {\begin{Vmatrix} \Gamma X \end{Vmatrix}}^2 $。

其中，Γ就是岭回归`Ridge`的`alpha`参数，指单位矩阵的倍数。上面的例子用的是默认值。我们可以看看岭回归参数：

```
Ridge()
```

```
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, solver='auto', tol=0.001)
```

岭回归相关系数的解是：

$$\beta = {(X^TX + \Gamma ^ T \Gamma)}^{-1}X \hat y$$

前面的一半和线性回归的相关系数的解是一样的，多了$\Gamma ^ T \Gamma)$一项。矩阵A的AA^T的结果是对称矩阵，且是半正定矩阵（对任意非0向量x，有$x^TAx \ge 0$）。相当于在线性回归的目标函数分母部分增加了一个很大的数。这样就把相关系数挤向0了。这样的解释比较粗糙，要深入了解，建议你看看SVD（矩阵奇异值分解）与岭回归的关系。

2.4 优化岭回归参数

当你使用岭回归模型进行建模时，需要考虑 `Ridge` 的 `alpha` 参数。

例如，用OLS（普通最小二乘法）做回归也许可以显示两个变量之间的某些关系；但是，当 `alpha` 参数正则化之后，那些关系就会消失。做决策时，这些关系是否需要考虑就显得很重要了。

Getting ready

这是我们第一个进行模型参数优化的主题，通常用交叉检验（cross validation）完成。在后面的主题中，还会有更简便的方式实现这些，但是这里我们一步一步来实现岭回归的优化。

在scikit-learn里面，岭回归的Γ参数就是 `RidgeRegression` 的 `alpha` 参数；因此，问题就是最优的 `alpha` 参数是什么。首先我们建立回归数据集：

```
from sklearn.datasets import make_regression
reg_data, reg_target = make_regression(n_samples=100, n_features=
2, effective_rank=1, noise=10)
```

How to do it...

在 `linear_models` 模块中，有一个对象叫 `RidgeCV` ，表示岭回归交叉检验（**ridge cross-validation**）。这个交叉检验类似于留一交叉验证法（**leave-one-out cross-validation**，**LOOCV**）。这种方法是指训练数据时留一个样本，测试的时候用这个未被训练过的样本：

```
import numpy as np
from sklearn.linear_model import RidgeCV
rcv = RidgeCV(alphas=np.array([.1, .2, .3, .4]))
rcv.fit(reg_data, reg_target)
```

```
RidgeCV(alphas=array([ 0.1,  0.2,  0.3,  0.4]), cv=None, fit_int
ercept=True,
    gcv_mode=None, normalize=False, scoring=None, store_cv_value
s=False)
```

拟合模型之后，`alpha` 参数就是最优参数：

```
rcv.alpha_
```

```
0.10000000000000001
```

这里，`0.1` 是最优参数，我们还想看到 `0.1` 附近更精确的值：

```
rcv = RidgeCV(alphas=np.array([.08, .09, .1, .11, .12]))
rcv.fit(reg_data, reg_target)
```

```
RidgeCV(alphas=array([ 0.08,  0.09,  0.1 ,  0.11,  0.12]), cv=No
ne,
    fit_intercept=True, gcv_mode=None, normalize=False, scoring=
None,
    store_cv_values=False)
```

```
rcv.alpha_
```

```
0.080000000000000002
```

可以按照这个思路一直优化下去，这里只做演示，后面还是介绍更好的方法。

How it works...

上面的演示很直接，但是我们介绍一下为什么这么做，以及哪个值才是最优的。在交叉检验的每一步里，模型的拟合效果都是用测试样本的误差表示。默认情况使用平方误差。更多细节见 `There's more...` 一节。

我们可以让 `RidgeCV` 储存交叉检验的数据，这样就可以可视化整个过程：

```
alphas_to_test = np.linspace(0.0001, 0.05)
rcv3 = RidgeCV(alphas=alphas_to_test, store_cv_values=True)
rcv3.fit(reg_data, reg_target)
```

```
RidgeCV(alphas=array([ 0.0001 ,  0.00112,  0.00214,  0.00316,  0
.00417,  0.00519,
        0.00621,  0.00723,  0.00825,  0.00927,  0.01028,  0.0113
 ,
        0.01232,  0.01334,  0.01436,  0.01538,  0.01639,  0.0174
1,
        0.01843,  0.01945,  0.02047,  0.02149,  0.0225 ,  0.0235
2,
        0.02454,  0.02556...4185,
        0.04287,  0.04389,  0.04491,  0.04593,  0.04694,  0.0479
6,
        0.04898,  0.05   ]),
    cv=None, fit_intercept=True, gcv_mode=None, normalize=False,
    scoring=None, store_cv_values=True)
```

你会看到，我们测试了0.0001到0.05区间中的50个点。由于我们把 `store_cv_values` 设置成 `true` ，我们可以看到每一个值对应的拟合效果：

```
rcv3.cv_values_.shape
```

```
(100, 50)
```

通过100个样本的回归数据集，我们获得了50个不同的 `alpha` 值。我们可以看到50个误差值，最小的均值误差对应最优的 `alpha` 值：

```
smallest_idx = rcv3.cv_values_.mean(axis=0).argmin()
alphas_to_test[smallest_idx]
```

```
0.014357142857142857
```

此时问题转化成了“RidgeCV认可我们的选择吗？”可以再用下面的命令获取 `alpha` 值：

```
rcv3.alpha_
```

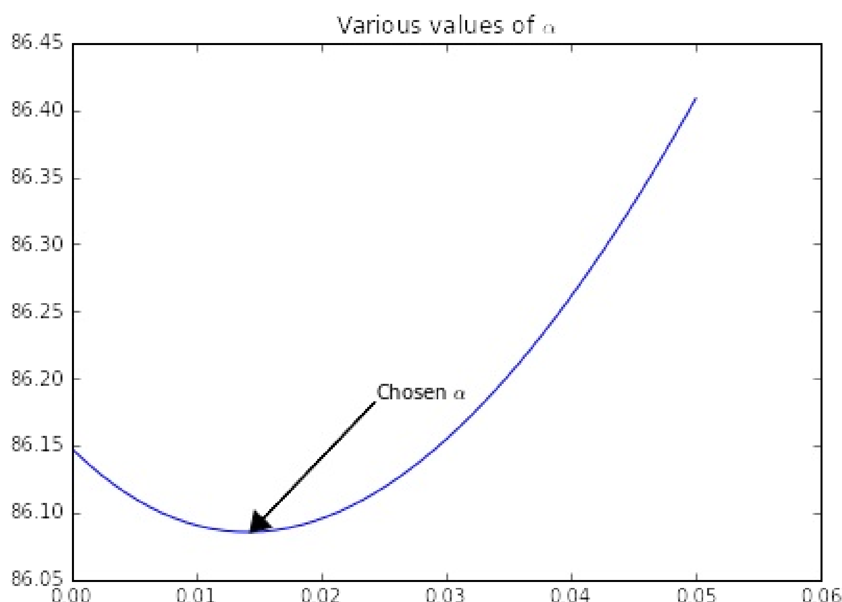
```
0.014357142857142857
```

通过可视化图形可以更直观的显示出来。我们画出50个测试 `alpha` 值的图：

```
%matplotlib inline
import matplotlib.pyplot as plt
f, ax = plt.subplots(figsize=(7, 5))
ax.set_title(r"Various values of $\alpha$")

xy = (alphas_to_test[smallest_idx], rcv3.cv_values_.mean(axis=0)
[smallest_idx])
xytext = (xy[0] + .01, xy[1] + .1)

ax.annotate(r'Chosen $\alpha$', xy=xy, xytext=xytext,
            arrowprops=dict(facecolor='black', shrink=0, width=0
)
            )
ax.plot(alphas_to_test, rcv3.cv_values_.mean(axis=0));
```



There's more...

如果我们想用其他误差自定义评分函数，也是可以实现的。前面我们介绍过MAD误差，我们可以用它来评分。首先我们需要定义损失函数：

```
def MAD(target, prediction):
    absolute_deviation = np.abs(target - prediction)
    return absolute_deviation.mean()
```

定义损失函数之后，我们用 `sklearn` 量度中的 `make_scorer` 函数来处理。这样做可以标准化自定义的函数，让scikit-learn对象可以使用它。另外，由于这是一个损失函数不是一个评分函数，是越低越好，所以要用 `sklearn` 来把最小化问题转化成最大化问题：

```
import sklearn
MAD = sklearn.metrics.make_scorer(MAD, greater_is_better=False)
rcv4 = RidgeCV(alphas=alphas_to_test, store_cv_values=True, scor
ing=MAD)
rcv4.fit(reg_data, reg_target)
smallest_idx = rcv4.cv_values_.mean(axis=0).argmin()
alphas_to_test[smallest_idx]
```

```
0.050000000000000003
```

2.5 LASSO正则化

LASSO（ least absolute shrinkage and selection operator，最小绝对值收缩和选择算子）方法与岭回归和LARS（least angle regression，最小角回归）很类似。与岭回归类似，它也是通过增加惩罚函数来判断、消除特征间的共线性。与LARS相似的是它也可以用作参数选择，通常得出一个相关系数的稀疏向量。

Getting ready

岭回归也不是万能药。有时就需要用LASSO回归来建模。本主题将用不同的损失函数，因此就要用对应的效果评估方法。

How to do it...

首先，我们还是用 `make_regression` 函数来建立数据集：

```
from sklearn.datasets import make_regression
reg_data, reg_target = make_regression(n_samples=200, n_features=
500, n_informative=5, noise=5)
```

之后，我们导入 `lasso` 对象：

```
from sklearn.linear_model import Lasso
lasso = Lasso()
```

`lasso` 包含很多参数，但是最意思的参数是 `alpha` ，用来调整 `lasso` 的惩罚项，在**How it works...**会具体介绍。现在我们用默认值 `1` 。另外，和岭回归类似，如果设置为 `0` ，那么 `lasso` 就是线性回归：

```
lasso.fit(reg_data, reg_target)
```

```
Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_sta
te=None,
   selection='cyclic', tol=0.0001, warm_start=False)
```

再让我们看看还有多少相关系数非零：

```
import numpy as np
np.sum(lasso.coef_ != 0)
```

```
9
```

```
lasso_0 = Lasso(0)
lasso_0.fit(reg_data, reg_target)
np.sum(lasso_0.coef_ != 0)
```

```
d:\programfiles\Miniconda3\lib\site-packages\IPython\kernel\__ma
in__.py:2: UserWarning: With alpha=0, this algorithm does not co
nverge well. You are advised to use the LinearRegression estimat
or
  from IPython.kernel.zmq import kernelapp as app
d:\programfiles\Miniconda3\lib\site-packages\sklearn\linear_mode
l\coordinate_descent.py:432: UserWarning: Coordinate descent wit
h alpha=0 may lead to unexpected results and is discouraged.
  positive)





500
```

和我们设想的一样，如果用线性回归，没有一个相关系数变成0。而且，如果你这么运行代码，scikit-learn会给出建议，就像上面显示的那样。

How it works...

对线性回归来说，我们是最小化残差平方和。而LASSO回归里，我们还是最小化残差平方和，但是加了一个惩罚项会导致稀疏。如下所示：

$$\sum {e_i + \lambda \ {\begin{Vmatrix} \beta \end{Vmatrix}}_1}

$$

最小化残差平方和的另一种表达方式是：

$$ RSS(\beta),其中 {\begin{Vmatrix} \beta \end{Vmatrix}}_1 \lt \beta

$$

这个约束会让数据稀疏。LASSO回归的约束创建了围绕原点的超立方体（相关系数是轴），也就意味着大多数点都在各个顶点上，那里相关系数为0。而岭回归创建的是超平面，因为其约束是L2范数，少一个约束，但是即使有限制相关系数也不会变成0。

LASSO交叉检验

上面的公式中，选择适当的λ（在scikit-learn的 `Lasso` 里面是 `alpha` ，但是书上都是λ）参数是关键。我们可以自己设置，也可以通过交叉检验来获取最优参数：

```
from sklearn.linear_model import LassoCV
lassocv = LassoCV()
lassocv.fit(reg_data, reg_target)
```

```
LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001, fit_interc
ept=True,
    max_iter=1000, n_alphas=100, n_jobs=1, normalize=False, posi
tive=False,
    precompute='auto', random_state=None, selection='cyclic', to
l=0.0001,
    verbose=False)
```

`lassocv` 有一个属性就是确定最合适的λ：

```
lassocv.alpha_
```

```
0.58535963603062136
```

计算的相关系数也可以看到：

```
lassocv.coef_[:5]
```

```
array([ 0.        , -0.        ,  0.        ,  0.0192606, -0.        
 ])
```

用最近的参数拟合后，`lassocv`的非零相关系数有29个：

```
np.sum(lassocv.coef_ != 0)
```

```
29
```

LASSO特征选择

LASSO通常用来为其他方法所特征选择。例如，你可能会用LASSO回归获取适当的特征变量，然后在其他算法中使用。

要获取想要的特征，需要创建一个非零相关系数的列向量，然后再其他算法拟合：

```
mask = lassocv.coef_ != 0
new_reg_data = reg_data[:, mask]
new_reg_data.shape
```

```
(200, 29)
```

2.6 LARS正则化

如果斯坦福大学的Bradley Efron, Trevor Hastie, Iain Johnstone和Robert Tibshirani没有发现它的话[1]，LARS(Least Angle Regression，最小角回归)可能有一天会被你想出来，它借用了威廉·吉尔伯特·斯特朗（William Gilbert Strang）介绍过的高斯消元法（Gaussian elimination）的灵感。

Getting ready

LARS是一种回归手段，适用于解决高维问题，也就是$p >> n$的情况，其中p表示列或者特征变量，n表示样本数量。

How to do it...

首先让我们导入必要的对象。这里我们用的数据集是200个数据，500个特征。我们还设置了一个低噪声，和少量提供信息的（informative）特征：

```
import numpy as np
from sklearn.datasets import make_regression
reg_data, reg_target = make_regression(n_samples=200,n_features=
500, n_informative=10, noise=2)
```

由于我们用了10个信息特征，因此我们还要为LARS设置10个非0的相关系数。我们事先可能不知道信息特征的准确数量，但是出于试验的目的是可行的：

```
from sklearn.linear_model import Lars
lars = Lars(n_nonzero_coefs=10)
lars.fit(reg_data, reg_target)
```

```
Lars(copy_X=True, eps=2.220446049250313e-16, fit_intercept=True
,
   fit_path=True, n_nonzero_coefs=10, normalize=True, precompute
='auto',
   verbose=False)
```

我们可以检验一下看看LARS的非0相关系数的和：

```
np.sum(lars.coef_ != 0)
```

```
10
```

问题在于为什么少量的特征反而变得更加有效。要证明这一点，让我们用一半数量来训练两个LARS模型，一个用12个非零相关系数，另一个非零相关系数用默认值。这里用12个是因为我们对重要特征的数量有个估计，但是可能无法确定准确的数量：

```
train_n = 100
lars_12 = Lars(n_nonzero_coefs=12)
lars_12.fit(reg_data[:train_n], reg_target[:train_n])
```

```
Lars(copy_X=True, eps=2.220446049250313le-16, fit_intercept=True
,
   fit_path=True, n_nonzero_coefs=12, normalize=True, precompute
='auto',
   verbose=False)
```

```
lars_500 = Lars() #默认就是500
lars_500.fit(reg_data[:train_n], reg_target[:train_n])
```

```
Lars(copy_X=True, eps=2.220446049250313le-16, fit_intercept=True
,
   fit_path=True, n_nonzero_coefs=500, normalize=True, precomput
e='auto',
   verbose=False)
```

现在，让我们看看拟合数据的效果如何，如下所示：

```
np.mean(np.power(reg_target[train_n:] - lars.predict(reg_data[tr
ain_n:]), 2))
```

```
18.607806437043894
```

```
np.mean(np.power(reg_target[train_n:] - lars_12.predict(reg_data
[train_n:]), 2))
```

```
529.97993250189643
```

```
np.mean(np.power(reg_target[train_n:] - lars_500.predict(reg_dat
a[train_n:]), 2))
```

```
2.3236770314162846e+34
```

仔细看看这组结果；测试集的误差明显高很多。高维数据集问题就在于此；通常面对大量的特征时，想找出一个对训练集拟合很好的模型并不难，但是拟合过度却是更大的问题。

How it works...

LARS通过重复选择与残存变化相关的特征。从图上看，相关性实际上就是特征与残差之间的最小角度；这就是LARS名称的由来。

选择第一个特征之后，LARS会继续沿着最小角的方向移动，直到另一个特征与残差有同样数量的相关性。然后，LARS会沿着两个特征组合的角度移动。如下图所示：

```
%matplotlib inline
import matplotlib.pyplot as plt
def unit(*args):
    squared = map(lambda x: x**2, args)
    distance = sum(squared) ** (.5)
    return map(lambda x: x / distance, args)

f, ax = plt.subplots(nrows=3, figsize=(5, 10))
plt.tight_layout()
ax[0].set_ylim(0, 1.1)
ax[0].set_xlim(0, 1.1)

x, y = unit(1, 0.02)
ax[0].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[0].text(x + .05, y + .05, r"$x_1$")

x, y = unit(.5, 1)
ax[0].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[0].text(x + .05, y + .05, r"$x_2$")

x, y = unit(1, .45)
ax[0].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[0].text(x + .05, y + .05, r"$y$")

ax[0].set_title("No steps")

# step 1
ax[1].set_title("Step 1")
ax[1].set_ylim(0, 1.1)
ax[1].set_xlim(0, 1.1)

x, y = unit(1, 0.02)
ax[1].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[1].text(x + .05, y + .05, r"$x_1$")

x, y = unit(.5, 1)
ax[1].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[1].text(x + .05, y + .05, r"$x_2$")

x, y = unit(.5, 1)
ax[1].arrow(.5, 0.01, x, y, ls='dashed', edgecolor='black', face
color='black')
```

```
ax[1].text(x + .5 + .05, y + .01 + .05, r"$x_2$")

ax[1].arrow(0, 0, .47, .01, width=.0015, edgecolor='black', face
color='black')
ax[1].text(.47-.15, .01 + .03, "Step 1")

x, y = unit(1, .45)
ax[1].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[1].text(x + .05, y + .05, r"$y$")

# step 2
ax[2].set_title("Step 2")
ax[2].set_ylim(0, 1.1)
ax[2].set_xlim(0, 1.1)

x, y = unit(1, 0.02)
ax[2].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[2].text(x + .05, y + .05, r"$x_1$")

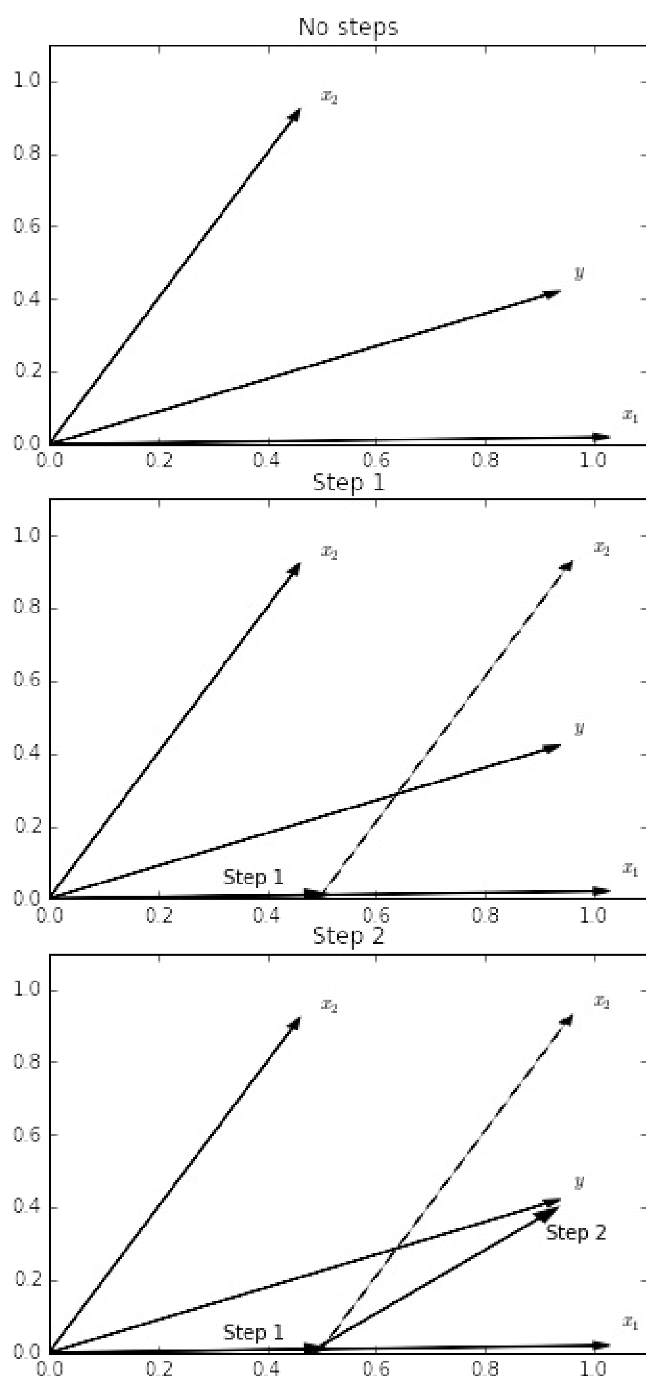
x, y = unit(.5, 1)
ax[2].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[2].text(x + .05, y + .05, r"$x_2$")

x, y = unit(.5, 1)
ax[2].arrow(.5, 0.01, x, y, ls='dashed', edgecolor='black', face
color='black')
ax[2].text(x + .5 + .05, y + .01 + .05, r"$x_2$")

ax[2].arrow(0, 0, .47, .01, width=.0015, edgecolor='black', face
color='black')
ax[2].text(.47-.15, .01 + .03, "Step 1")

##  step 2
x, y = unit(1, .45)
ax[2].arrow(.5, .02, .4, .35, width=.0015, edgecolor='black', fa
cecolor='black')
ax[2].text(x, y - .1, "Step 2")

x, y = unit(1, .45)
ax[2].arrow(0, 0, x, y, edgecolor='black', facecolor='black')
ax[2].text(x + .05, y + .05, r"$y$");
```



具体过程是，我们把$x2$沿着$x1$方向移动到一个位置：$x1$与y的点积与$x1$与y的点积相同。到了这个位置之后，我们再沿着$x1$和$x2$夹角的一半的方向移动。

There's more...

和我们前面用交叉检验来优化领回归模型一样，我们可以对LARS做交叉检验：

```
from sklearn.linear_model import LarsCV
lcv = LarsCV()
lcv.fit(reg_data, reg_target)
```

```
d:\Miniconda3\lib\site-packages\sklearn\linear_model\least_angle
.py:285: ConvergenceWarning: Regressors in active set degenerate
. Dropping a regressor, after 168 iterations, i.e. alpha=2.278e-
02, with an active set of 132 regressors, and the smallest chole
sky pivot element being 6.144e-08
  ConvergenceWarning)
d:\Miniconda3\lib\site-packages\sklearn\linear_model\least_angle
.py:285: ConvergenceWarning: Regressors in active set degenerate
. Dropping a regressor, after 168 iterations, i.e. alpha=2.105e-
02, with an active set of 132 regressors, and the smallest chole
sky pivot element being 9.771e-08
  ConvergenceWarning)





LarsCV(copy_X=True, cv=None, eps=2.220446049250313e-16, fit_int
ercept=True,
    max_iter=500, max_n_alphas=1000, n_jobs=1, normalize=True,
    precompute='auto', verbose=False)
```

用交叉检验可以帮助我们确定需要使用的非零相关系数的最佳数量。验证如下所示：

```
np.sum(lcv.coef_ != 0)
```

```
43
```

说实话，LARS的精髓还没有领会，抽空会把原文译出来，看各种解释不如看原文。

[1] Efron, Bradley; Hastie, Trevor; Johnstone, Iain and Tibshirani, Robert(2004). "Least Angle Regression". Annals of Statistics 32(2): pp. 407–499.doi:10.1214/009053604000000067. MR 2060166.

2.7 用线性方法处理分类问题——逻辑回归

实际上线性模型也可以用于分类任务。方法是把一个线性模型拟合成某个类型的概率分布，然后用一个函数建立阈值来确定结果属于哪一类。

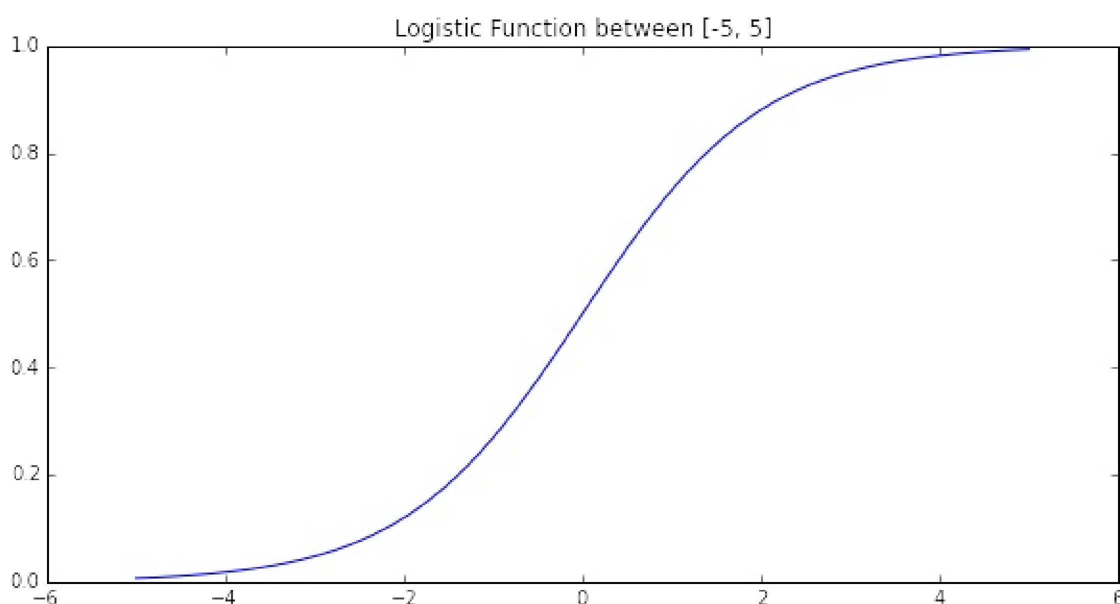
Getting ready

这里用的函数是经典的逻辑函数。一个非常简单的函数：

$$f(x)= \frac 1 {1+e^{-t}}$$

它的图形如下图所示：

```
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
f, ax = plt.subplots(figsize=(10, 5))
rng = np.linspace(-5, 5)
log_f = np.apply_along_axis(lambda x:1 / (1 + np.exp(-x)), 0, rn
g)
ax.set_title("Logistic Function between [-5, 5]")
ax.plot(rng, log_f);
```



让我们用 `make_classification` 方法创建一个数据集来进行分类：

```
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4)
```

How to do it...

`LogisticRegression` 对象和其他线性模型的用法一样：

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
```

我们将把前面200个数据作为训练集，最后200个数据作为测试集。因为这是随机数据集，所以用最后200个数据没问题。但是如果处理具有某种结构的数据，就不能这么做了（例如，你的数据集是时间序列数据）：

```
X_train = X[:-200]
X_test = X[-200:]
y_train = y[:-200]
y_test = y[-200:]
```

在本书后面的内容里，我们将详细介绍交叉检验。这里，我们需要的只是用逻辑回归拟合模型。我们会关注训练集的预测结果，就像测试集预测结果一样。经常对比两个数据集预测正确率是个好做法。通常，你在训练集获得的结果更好；模型在测试集上预测失败的比例也至关重要：

```
lr.fit(X_train, y_train)
y_train_predictions = lr.predict(X_train)
y_test_predictions = lr.predict(X_test)
```

现在我们有了预测值，让我们看看预测的效果。这里，我们只简单看看预测正确的比例；后面，我们会详细的介绍分类模型效果的评估方法。

计算很简单，就是用预测正确的数量除以总样本数：

```
(y_train_predictions == y_train).sum().astype(float) / y_train.shape[0]
```

```
0.89375000000000004
```

测试集的效果是：

```
(y_test_predictions == y_test).sum().astype(float) / y_test.shape[0]
```

```
0.90500000000000003
```

可以看到，测试集的正确率和训练集的结果差不多。但是实际中通常差别很大。

现在问题变成，怎么把逻辑函数转换成分类方法。

首先，线性回归希望找到一个线性方程拟合出给定自变量X条件下因变量Y的期望值，就是$E(Y|X)=x \beta$。这里Y的值是某个类型发生的概率。因此，我们要解决的分类问题就是$E(p|X)=x \beta$。然后，只要阈值确定，就会有$Logit(p) = X \beta$。这个理念的扩展形式可以构成许多形式的回归行为，例如，泊松过程（Poisson）。

There's more...

下面的内容你以后肯定会遇到。一种情况是一个类型与其他类型的权重不同；例如，一个能可能权重很大，99%。这种情况在分类工作中经常遇到。经典案例就是信用卡虚假交易检测，大多数交易都不是虚假交易，但是不同类型误判的成本相差很大。

让我们建立一个分类问题，类型y的不平衡权重95%，我们看看基本的逻辑回归模型如何处理这类问题：

```
X, y = make_classification(n_samples=5000, n_features=4, weights
=[.95])
sum(y) / (len(y)*1.) #检查不平衡的类型
```

```
0.0562
```

建立训练集和测试集，然后用逻辑回归拟合：

```
X_train = X[:-500]
X_test = X[-500:]
y_train = y[:-500]
y_test = y[-500:]

lr.fit(X_train, y_train)
y_train_predictions = lr.predict(X_train)
y_test_predictions = lr.predict(X_test)
```

现在我们在看看模型拟合的情况：

```
(y_train_predictions == y_train).sum().astype(float) / y_train.s
hape[0]
```

```
0.9671111111111114
```

```
(y_test_predictions == y_test).sum().astype(float) / y_test.shap
e[0]
```

```
0.9679999999999997
```

结果看着还不错，但这是说如果我们把一个交易预测成正常交易（或者称为类型0），那么我们有95%左右的可能猜对。如果我们想看看模型对类型1的预测情况，可能就不是那么好了：

```
(y_test[y_test==1] == y_test_predictions[y_test==1]).sum().astyp
e(float) / y_test[y_test==1].shape[0]
```

```
0.5
```

如果相比正常交易，我们更关心虚假交易；那么这是由商业规则决定的，我们可能会改变预测正确和预测错误的权重。

通常情况下，虚假交易与正常交易的权重与训练集的类型权重的倒数一致。但是，因为我们更关心虚假交易，所有让我们用多重采样（oversample）方法来表示虚假交易与正常交易的权重。

```
lr = LogisticRegression(class_weight={0: .15, 1: .85})
lr.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight={0: 0.15, 1: 0.85}, dual=
False,
          fit_intercept=True, intercept_scaling=1, max_iter=100,
          multi_class='ovr', penalty='l2', random_state=None,
          solver='liblinear', tol=0.0001, verbose=0)
```

让我们再预测一下结果：

```
y_train_predictions = lr.predict(X_train)
y_test_predictions = lr.predict(X_test)
```

```
(y_test[y_test==1] == y_test_predictions[y_test==1]).sum().astyp
e(float) / y_test[y_test==1].shape[0]
```

```
0.708333333333333337
```

但是，这么做需要付出什么代价？让我们看看：

```
(y_test_predictions == y_test).sum().astype(float) / y_test.shap
e[0]
```

```
0.9399999999999995
```

可以看到，准确率降低了3%。这样是否合适由你的问题决定。如果与虚假交易相关的评估成本非常高，那么它就能抵消追踪虚假交易付出的成本。

2.8 贝叶斯岭回归

在用岭回归弥补线性回归的不足主题中，我们介绍了岭回归优化的限制条件。我们还介绍了相关系数的先验概率分布的贝叶斯解释，将很大程度地影响着先验概率分布，先验概率分布通常均值是0。

因此，现在我们就来演示如何scikit-learn来应用这种解释。

Getting ready

岭回归和套索回归（lasso regression）用贝叶斯观点来解释，与频率优化观点解释相反。scikit-learn只实现了贝叶斯岭回归，但是在*How it works...*一节，我们将对比两种回归算法。

首先，我们创建一个回归数据集：

```
from sklearn.datasets import make_regression
X, y = make_regression(1000, 10, n_informative=2, noise=20)
```

How to do it...

我们可以把岭回归加载进来拟合模型：

```
from sklearn.linear_model import BayesianRidge
br = BayesianRidge()
```

有两组相关系数，分别是`alpha_1 / alpha_2`和`lambda_1 / lambda_2`。其中，`alpha_*`是先验概率分布的α超参数，`lambda_*`是先验概率分布的λ超参数。

首先，让我们不调整参数直接拟合模型：

```
br.fit(X, y)
br.coef_
```

```
array([ -1.39241213,   0.14671513,  -0.08150797,  37.50250891,
         0.21850082,  -0.78482779,  -0.26717555,  -0.71319956,
         0.7926308 ,   5.74658302])
```

现在，我们来调整超参数，注意观察相关系数的变化：

```
br_alphas = BayesianRidge(alpha_1=10, lambda_1=10)
br_alphas.fit(X, y)
br_alphas.coef_
```

```
array([ -1.38807423,   0.14050794,  -0.08309391,  37.3032803 ,
         0.2254332 ,  -0.77031801,  -0.27005478,  -0.71632657,
         0.78501276,   5.71928608])
```

How it works...

因为是贝叶斯岭回归，我们假设先验概率分布带有误差和α参数，先验概率分布都服从Γ分布。

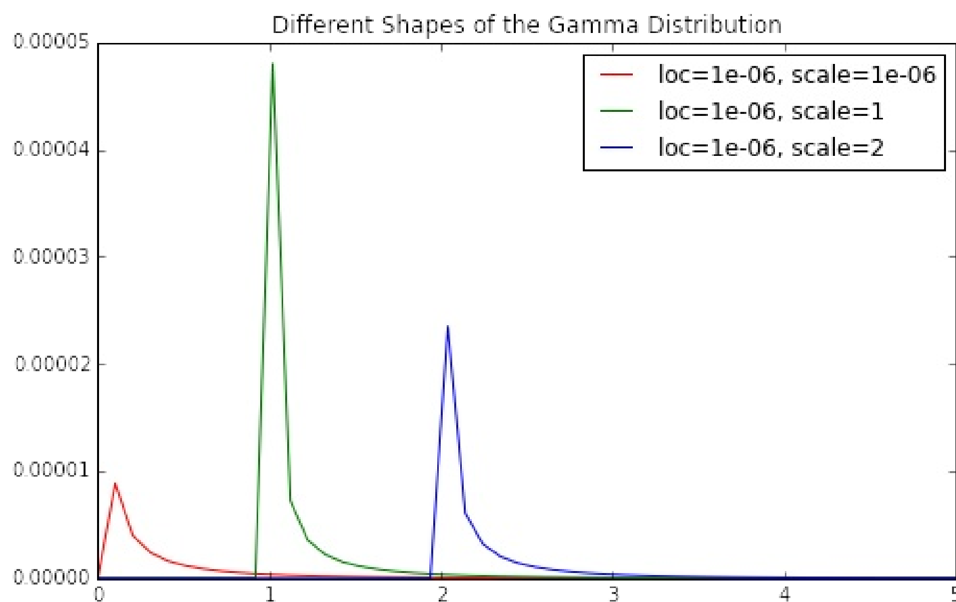
Γ分布是一种极具灵活性的分布。不同的形状参数和尺度参数的Γ分布形状有差异。**1e-06**是 scikit-learn里面`BayesianRidge`形状参数的默认参数值。

```
import matplotlib.pyplot as plt
%matplotlib inline

from scipy.stats import gamma
import numpy as np

form = lambda x, y: "loc={}, scale={}".format(x, y)
g = lambda x, y=1e-06, z=1e-06: gamma.pdf(x, y, z)
g2 = lambda x, y=1e-06, z=1: gamma.pdf(x, y, z)
g3 = lambda x, y=1e-06, z=2: gamma.pdf(x, y, z)
rng = np.linspace(0, 5)
f, ax = plt.subplots(figsize=(8, 5))

ax.plot(rng, list(map(g, rng)), label=form(1e-06, 1e-06), color=
'r')
ax.plot(rng, list(map(g2, rng)), label=form(1e-06, 1), color='g'
)
ax.plot(rng, list(map(g3, rng)), label=form(1e-06, 2), color='b'
)
ax.set_title("Different Shapes of the Gamma Distribution")
ax.legend();
```



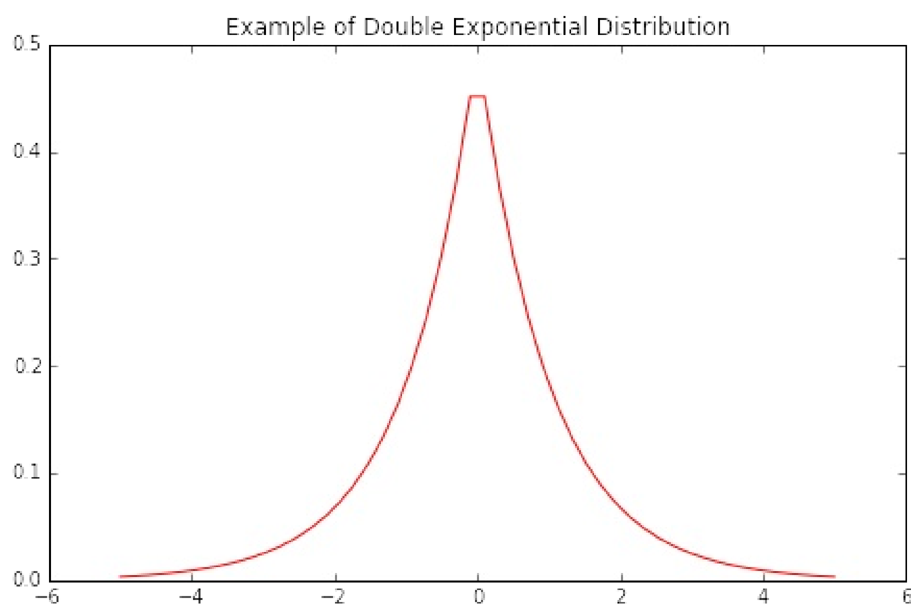
你会看到，相关系数最终都会收缩到0，尤其当形状参数特别小的时候。

There's more...

就像我前面介绍的，还有一种套索回归的贝叶斯解释。我们把先验概率分布看出是相关系数的函数；它们本身都是随机数。对于套索回归，我们选择一个可以产生0的分布，比如双指数分布（Double Exponential Distribution，也叫Laplace distribution）。

```
from scipy.stats import laplace
form = lambda x, y: "loc={}, scale={}".format(x, y)
g = lambda x: laplace.pdf(x)
rng = np.linspace(-5, 5)
f, ax = plt.subplots(figsize=(8, 5))

ax.plot(rng, list(map(g, rng)), color='r')
ax.set_title("Example of Double Exponential Distribution");
```



留意看x轴为0处的顶点。这将会使套索回归的相关系数为0。通过调整超参数，还有可能创建出相关系数为0的情况，这由问题的具体情况决定。

2.9 用梯度提升回归从误差中学习

梯度提升回归（Gradient boosting regression，GBR）是一种从它的错误中进行学习的技术。它本质上就是集思广益，集成一堆较差的学习算法进行学习。有两点需要注意：

- 每个学习算法准备率都不高，但是它们集成起来可以获得很好的准确率。
- 这些学习算法依次应用，也就是说每个学习算法都是在前一个学习算法的错误中学习

Getting ready

我们还是用基本的回归数据来演示GBR：

```
import numpy as np
from sklearn.datasets import make_regression
X, y = make_regression(1000, 2, noise=10)
```

How to do it...

GBR算是一种集成模型因为它是一个集成学习算法。这种称谓的含义是指GBR用许多较差的学习算法组成了一个更强大的学习算法：

```
from sklearn.ensemble import GradientBoostingRegressor as GBR
gbr = GBR()
gbr.fit(X, y)
gbr_preds = gbr.predict(X)
```

很明显，这里应该不止一个模型，但是这种模式现在很简明。现在，让我们用基本回归算法来拟合数据当作参照：

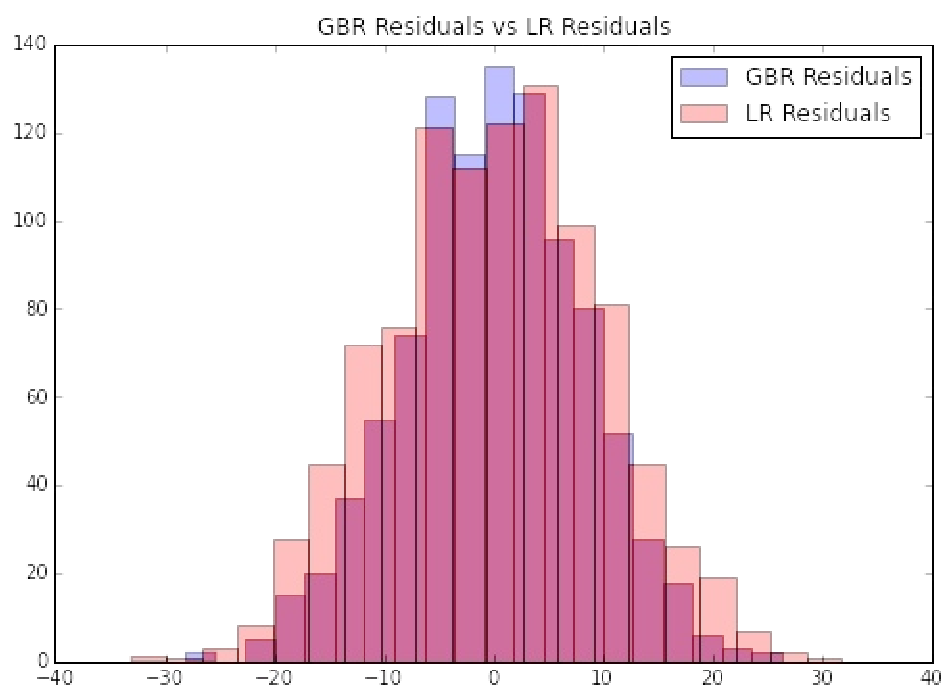
```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X, y)
lr_preds = lr.predict(X)
```

有了参照之后，让我们看看GBR算法与线性回归算法效果的对比情况。图像生成可以参照第一章正态随机过程的相关主题，首先需要下面的计算：

```
gbr_residuals = y - gbr_preds
lr_residuals = y - lr_preds
```

```
%matplotlib inline
from matplotlib import pyplot as plt
```

```
f, ax = plt.subplots(figsize=(7, 5))
f.tight_layout()
ax.hist(gbr_residuals,bins=20,label='GBR Residuals', color='b',
alpha=.5);
ax.hist(lr_residuals,bins=20,label='LR Residuals', color='r', al
pha=.5);
ax.set_title("GBR Residuals vs LR Residuals")
ax.legend(loc='best');
```



看起来好像GBR拟合的更好，但是并不明显。让我们用95%置信区间（Confidence interval,CI）对比一下：

```
np.percentile(gbr_residuals, [2.5, 97.5])
```

```
array([-16.73937398,  15.96258406])
```

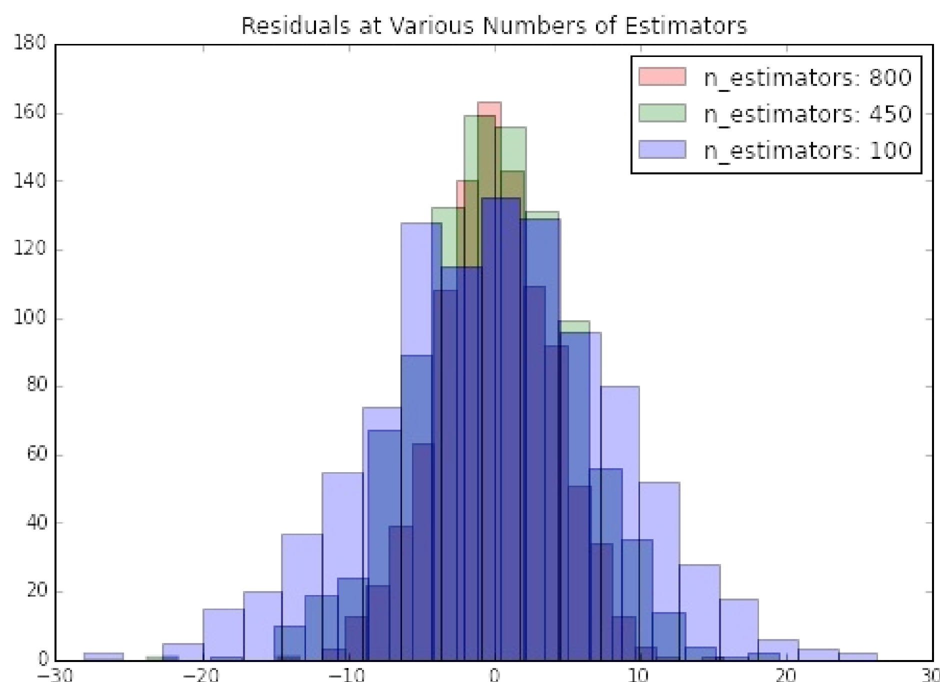
```
np.percentile(lr_residuals, [2.5, 97.5])
```

```
array([-19.03378242,  19.45950191])
```

GBR的置信区间更小，数据更集中，因此其拟合效果更好；我们还可以对GBR算法进行一些调整来改善效果。我用下面的例子演示一下，然后在下一节介绍优化方法：

```
n_estimators = np.arange(100, 1100, 350)
gbrs = [GBR(n_estimators=n_estimator) for n_estimator in n_estim
ators]
residuals = {}
for i, gbr in enumerate(gbrs):
    gbr.fit(X, y)
    residuals[gbr.n_estimators] = y - gbr.predict(X)
```

```
f, ax = plt.subplots(figsize=(7, 5))
f.tight_layout()
colors = {800:'r', 450:'g', 100:'b'}
for k, v in residuals.items():
    ax.hist(v,bins=20,label='n_estimators: %d' % k, color=colors
[k], alpha=.5);
ax.set_title("Residuals at Various Numbers of Estimators")
ax.legend(loc='best');
```



图像看着有点混乱，但是依然可以看出随着估计器数据的增加，误差在减少。不过，这并不是一成不变的。首先，我们没有交叉检验过，其次，随着估计器数量的增加，训练时间也会变长。现在我们用数据比较小没什么关系，但是如果数据再放大一两倍问题就出来了。

How it works...

上面例子中GBR的第一个参数是`n_estimators`，指GBR使用的学习算法的数量。通常，如果你的设备性能更好，可以把`n_estimators`设置的更大，效果也会更好。还有另外几个参数要说明一下。

你应该在优化其他参数之前先调整`max_depth`参数。因为每个学习算法都是一颗决策树，`max_depth`决定了树生成的节点数。选择合适的节点数量可以更好的拟合数据，而更多的节点数可能造成拟合过度。

`loss`参数决定损失函数，也直接影响误差。`ls`是默认值，表示最小二乘法（least squares）。还有最小绝对值差值，Huber损失和分位数损失（quantiles）等等。

第三章 使用距离向量构建模型

> 作者：Trent Hauck
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

这一章中，我们会涉及到聚类。聚类通常和非监督技巧组合到一起。这些技巧假设我们不知道结果变量。这会使结果模糊，以及实践客观。但是，聚类十分有用。我们会看到，我们可以使用聚类，将我们的估计在监督设置中“本地化”。这可能就是聚类非常高效的原因。它可以处理很大范围的情况，通常，结果也不怎么正常。

这一章中我们会浏览大量应用，从图像处理到回归以及离群点检测。通过这些应用，我们会看到聚类通常可以通过概率或者优化结构来观察。不同的解释会导致不同的权衡。我们会看到，如何训练模型，以便让工具尝试不同模型，在面对聚类问题的时候。

3.1 使用 KMeans 对数据聚类

聚类是个非常实用的技巧。通常，我们在采取行动时需要分治。考虑公司的潜在客户列表。公司可能需要将客户按类型分组，之后为这些分组划分职责。聚类可以使这个过程变得容易。

KMeans 可能是最知名的聚类算法之一，并且也是最知名的无监督学习技巧之一。

准备

首先，让我们看一个非常简单的聚类，之后我们再讨论 KMeans 如何工作。

```
>>> from sklearn.datasets import make_blobs
>>> blobs, classes = make_blobs(500, centers=3)
```

同样，由于我们绘制一些图表，导入 `matplotlib` ，像这样：

```
>>> import matplotlib.pyplot as plt
```

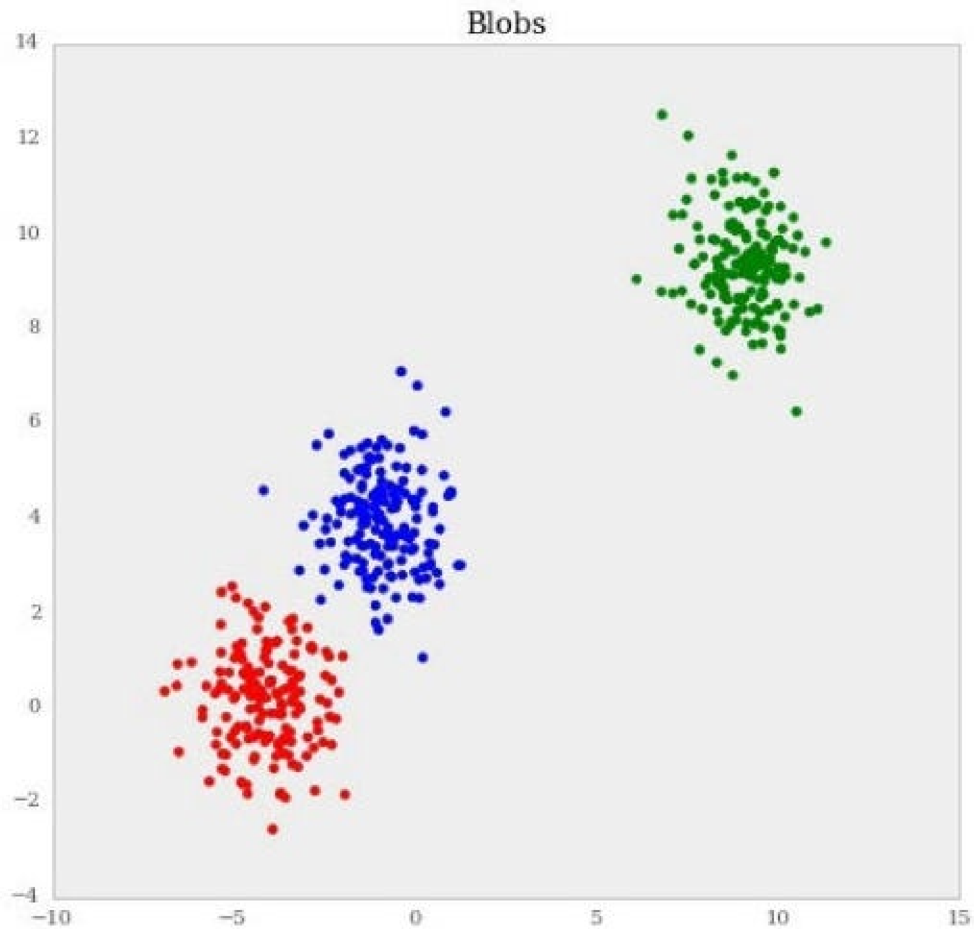
操作步骤

我们打算浏览一个简单的例子，它对伪造数据进行聚类。之后我们会稍微谈论一下，KMeans 如何工作，来寻找最优的块数量。

看一看我们的数据块，我们可以看到，有三个不同的簇。

```
>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])
>>> rgb = np.array(['r', 'g', 'b'])
>>> ax.set_title("Blobs")
```

输出如下：



现在我们可以使用 KMeans 来寻找这些簇的形心。第一个例子中，我们假装知道有三个形心。

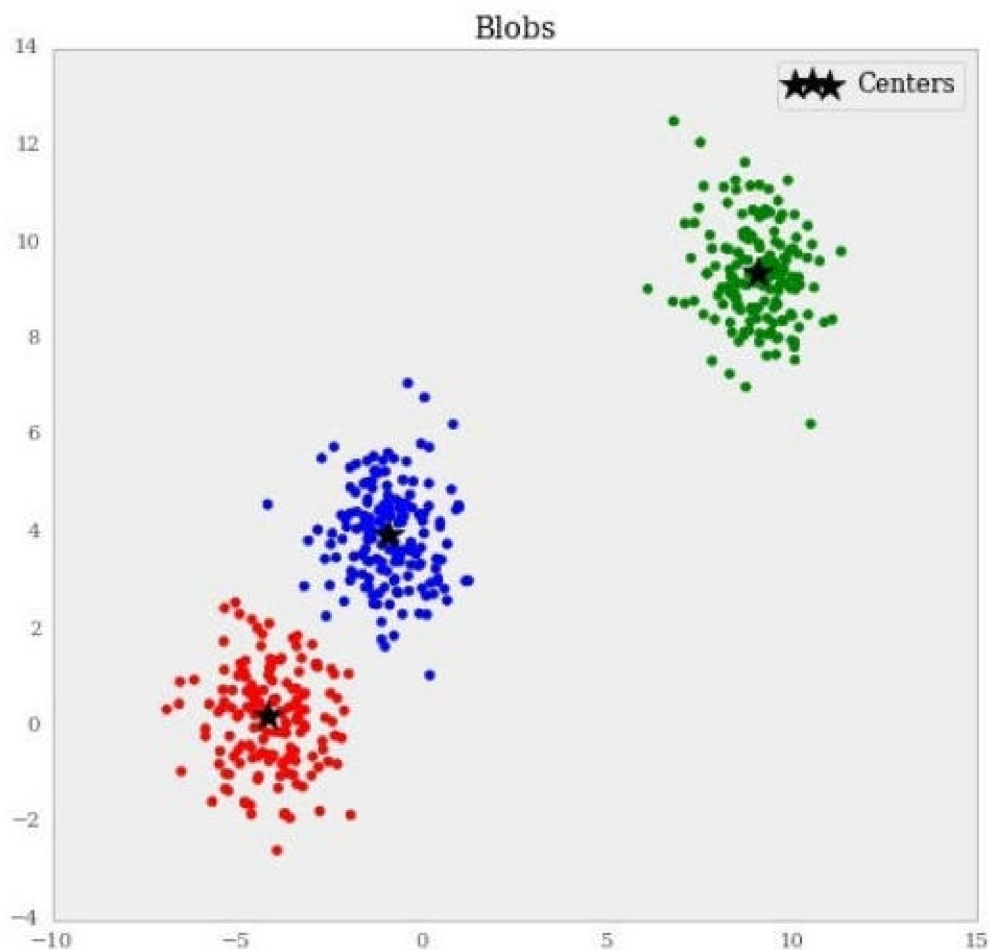
```
>>> from sklearn.cluster import KMeans
>>> kmean = KMeans(n_clusters=3)
>>> kmean.fit(blobs)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3
,
       n_init=10, n_jobs=1, precompute_distances=True,
       random_state=None, tol=0.0001, verbose=0)

>>> kmean.cluster_centers_ array([[ 0.47819567,  1.80819197],

[ 0.08627847,  8.24102715],
[ 5.2026125 ,  7.86881767]])

>>> f, ax = plt.subplots(figsize=(7.5, 7.5))
>>> ax.scatter(blobs[:, 0], blobs[:, 1], color=rgb[classes])
>>> ax.scatter(kmean.cluster_centers_[:, 0],
               kmean.cluster_centers_[:, 1], marker='*', s=250,
               color='black', label='Centers')
>>> ax.set_title("Blobs")
>>> ax.legend(loc='best')
```

下面的截图展示了输出：



其它属性也很实用。例如，`labels_` 属性会产生每个点的预期标签。

```
>>> kmean.labels_[:5]
array([1, 1, 2, 2, 1], dtype=int32)
```

我们可以检查，例如，`labels_` 是否和类别相同，但是由于 KMeans 不知道类别是什么，它不能给两个类别分配相同的索引值：

```
>>> classes[:5]
array([0, 0, 2, 2, 0])
```

将类别中的 `1` 变成 `0` 来查看是否与 `labels_` 匹配。

`transform` 函数十分有用，它会输出每个点到形心的距离。

```
>>> kmean.transform(blobs)[:5]
array([[ 6.47297373,  1.39043536,  6.4936008 ],
       [ 6.78947843,  1.51914705,  3.67659072],
       [ 7.24414567,  5.42840092,  0.76940367],
       [ 8.56306214,  5.78156881,  0.89062961],
       [ 7.32149254,  0.89737788,  5.12246797]])
```

工作原理

KMeans 实际上是个非常简单的算法，它使簇中的点到均值的距离的平方和最小。

首先它会设置一个预定义的簇数量 `K`，之后执行这些事情：

- 将每个数据点分配到最近的簇中。
- 通过计算初中每个数据点的均值，更新每个形心。

直到满足特定条件。

3.2 优化形心数量

形心难以解释，并且也难以判断是否数量正确。理解你的数据是否是未分类的十分重要，因为这会直接影响我们可用的评估手段。

准备

为无监督学习评估模型表现是个挑战。所以，在了解真实情况的时候，`sklearn` 拥有多种方式来评估聚类，但在不了解时就很少。

我们会以一个简单的簇模型开始，并评估它的相似性。这更多是出于机制的目的，因为测量一个簇的相似性在寻找簇数量的真实情况时显然没有用。

操作步骤

为了开始，我们会创建多个数据块，它们可用于模拟数据簇。

```
>>> from sklearn.datasets import make_blobs
>>> import numpy as np
>>> blobs, classes = make_blobs(500, centers=3)

>>> from sklearn.cluster import KMeans
>>> kmean = KMeans(n_clusters=3)
>>> kmean.fit(blobs)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3
,
       n_init=10, n_jobs=1, precompute_distances=True,
       random_state=None, tol=0.0001, verbose=0)
```

首先，我们查看轮廓（Silhouette）距离。轮廓距离是簇内不相似性、最近的簇间不相似性、以及这两个值最大值的比值。它可以看做簇间分离程度的度量。

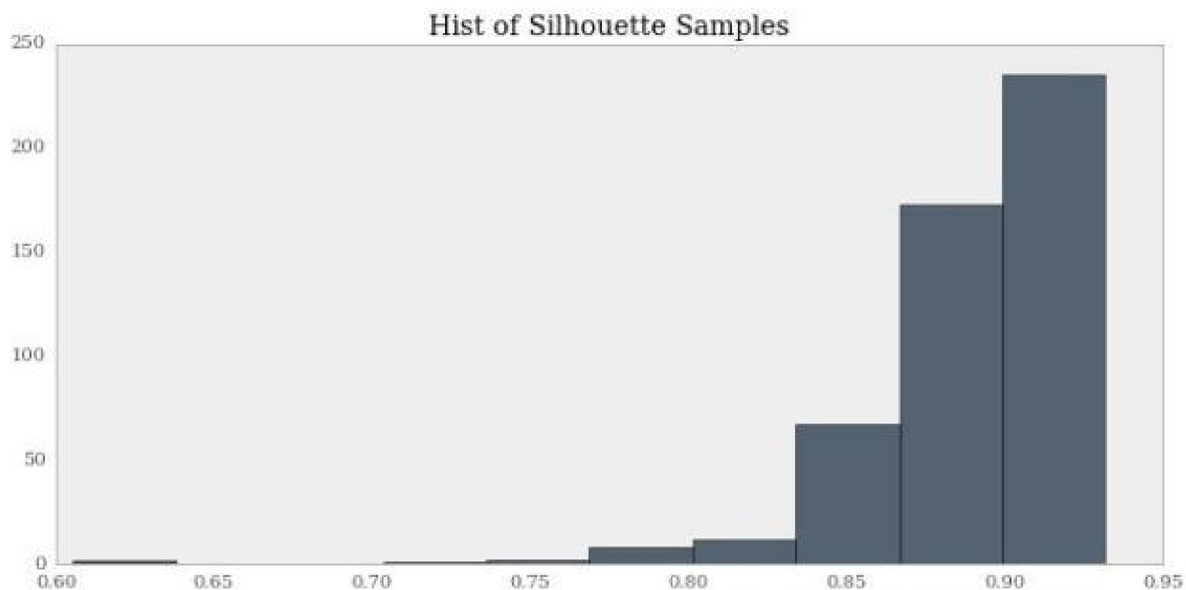
让我们看一看数据点到形心的距离分布，理解轮廓距离非常有用。

```
>>> from sklearn import metrics
>>> silhouette_samples = metrics.silhouette_samples(blobs,
                         kmean.labels_)
>>> np.column_stack((classes[:5], silhouette_samples[:5]))

array([[ 1.,  0.87617292],
       [ 1.,  0.89082363],
       [ 1.,  0.88544994],
       [ 1.,  0.91478369],
       [ 1.,  0.91308287]])
>>> f, ax = plt.subplots(figsize=(10, 5))

>>> ax.set_title("Hist of Silhouette Samples")
>>> ax.hist(silhouette_samples)
```

输出如下：



要注意，通常接近 1 的系数越高，分数就越高。

工作原理

轮廓系数的均值通常用于描述整个模型的拟合度。

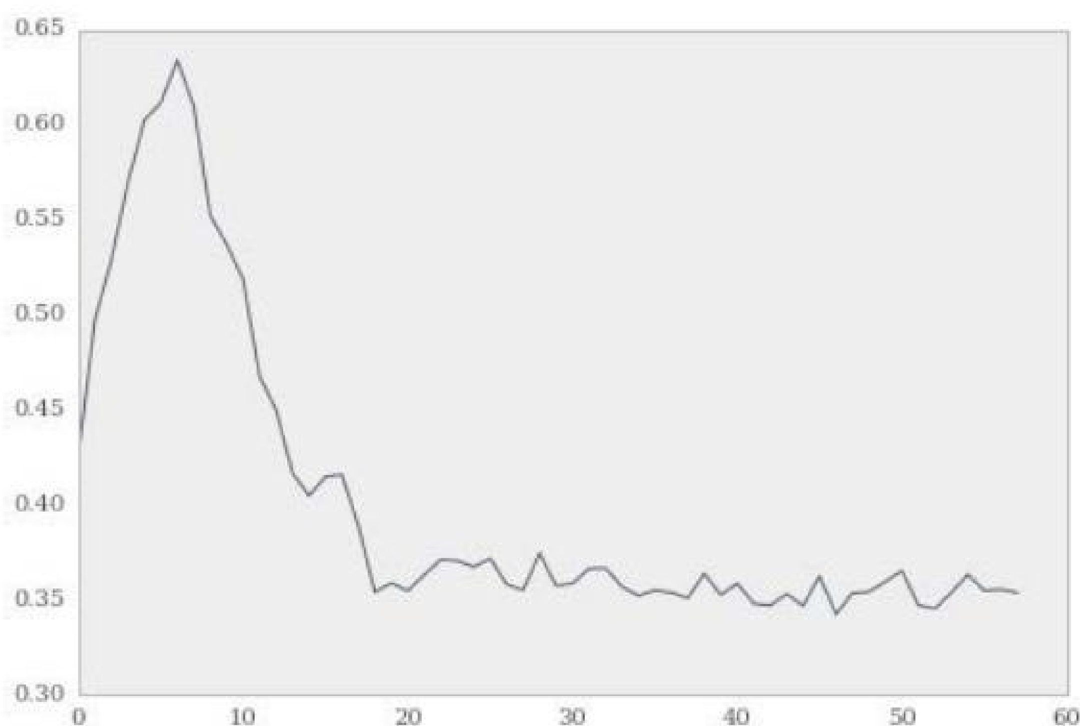
```
>>> silhouette_samples.mean()
0.5713046295333957８
```

这十分普遍，事实上， `metrics` 模块提供了一个函数来获得刚才的值。

现在，让我们训练多个簇的模型，并看看平均得分是什么样：

```
# first new ground truth
>>> blobs, classes = make_blobs(500, centers=10)
>>> sillhouette_avgs = []
# this could take a while
>>> for k in range(2, 60):
       kmean = KMeans(n_clusters=k).fit(blobs)
       sillhouette_avgs.append(metrics.silhouette_score(blobs,
                                       kmean.labels_))
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.plot(sillhouette_avgs)
```

下面是输出：



这个绘图表明，轮廓均值随着形心数量的变化情况。我们可以看到最优的数量是3，根据所生成的数据。但是最优的数量看起来是 6 或者 7。这就是聚类的实际情况，十分普遍，我们不能获得正确的簇数量，我们只能估计簇数量的近似值。

3.3 评估聚类的正确性

我们之前讨论了不知道真实情况的条件下的聚类评估。但是，我们还没有讨论簇已知条件下的 KMeans 评估。在许多情况下，这都是不可知的，但是如果存在外部的标注，我们就会知道真实情况，或者至少是代理。

准备

所以，让我们假设有一个世界，其中我们有一些外部代理，向我们提供了真实情况。

我们会创建一个简单的数据集，使用多种方式评估相对于真实庆康的正确性。之后讨论它们。

操作步骤

在我们开始度量之前，让我们先查看数据集：

```
>>> f, ax = plt.subplots(figsize=(7, 5))

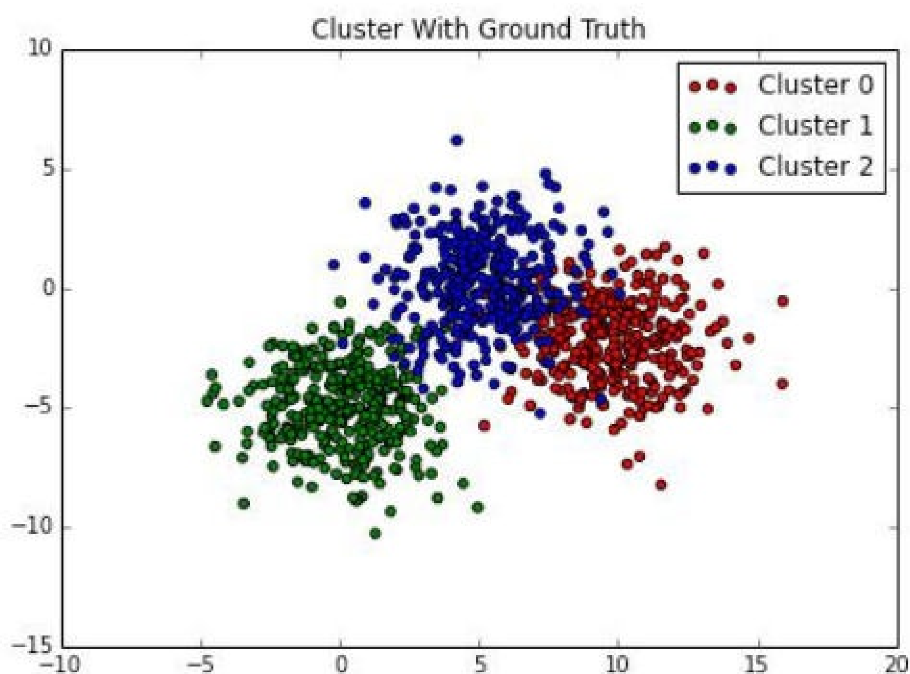
>>> colors = ['r', 'g', 'b']

>>> for i in range(3):
        p = blobs[ground_truth == i]
        ax.scatter(p[:,0], p[:,1], c=colors[i],
        label="Cluster {}".format(i))

>>> ax.set_title("Cluster With Ground Truth")
>>> ax.legend()

>>> f.savefig("94850S_03-16")
```

下面是输出：



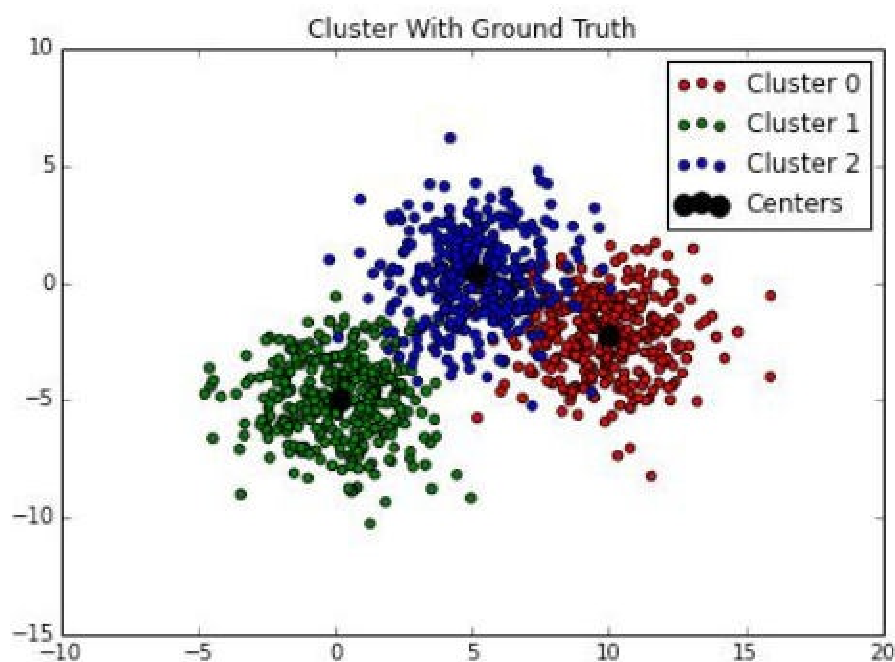
既然我们已经训练了模型，让我们看看簇的形心：

```
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> colors = ['r', 'g', 'b']

>>> for i in range(3):
       p = blobs[ground_truth == i]
       ax.scatter(p[:,0], p[:,1], c=colors[i],        label="Clu
ster {}".format(i))
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
               kmeans.cluster_centers_[:, 1], s=100,
               color='black',
               label='Centers')
>>> ax.set_title("Cluster With Ground Truth")
>>> ax.legend()
>>> f.savefig("94850S_03-17")
```

下面是输出：



既然我们能够将聚类表现看做分类练习，在其语境中有用的方法在这里也有用：

```
>>> for i in range(3):
       print (kmeans.labels_ == ground_truth)[ground_truth == i]
       .astype(int).mean()

0.0778443113772
0.990990990991
0.0570570570571
```

很显然我们有一些错乱的簇。所以让我们将其捋直，之后我们查看准确度。

```
>>> new_ground_truth = ground_truth.copy()
>>> new_ground_truth[ground_truth == 0] = 2
>>> new_ground_truth[ground_truth == 2] = 0

>>> for i in range(3):
       print (kmeans.labels_ == new_ground_truth)[ground_truth =
= i]
       .astype(int).mean()

0.919161676647
0.990990990991
0.90990990991
```

所以我们 90% 的情况下都是正确的。第二个相似性度量是互信息（ mutual information score）得分。

```
>>> from sklearn import metrics

>>> metrics.normalized_mutual_info_score(ground_truth, kmeans.la
bels_)

0.78533737204433651
```

分数靠近 0，就说明标签的分配可能不是按照相似过程生成的。但是分数靠近 1，就说明两个标签有很强的一致性。

例如，让我们看一看互信息分数自身的情况：

```
>>> metrics.normalized_mutual_info_score(ground_truth, ground_tr
uth)

1.0
```

通过名称，我们可以分辨出可能存在未规范化的 `mutual_info_score` ：

```
>>> metrics.mutual_info_score(ground_truth, kmeans.labels_)

0.78945287371677486
```

这非常接近了。但是，规范化的互信息是互信息除以每个真实值和标签的熵的乘积的平方根。

更多

有一个度量方式我们尚未讨论，并且不依赖于真实情况，就是惯性（inertia）度量。当前，它作为一种度量并没有详细记录。但是，它是 KMeans 中最简单的度量。

惯性是每个数据点和它所分配的簇的平方差之和。我们可以稍微使用 NumPy 来计算它：

```
>>> kmeans.inertia_
```

3.4 使用 MiniBatch KMeans 处理更多数据

KMeans 是一个不错的方法，但是不适用于大量数据。这是因为 KMenas 的复杂度。也就是说，我们可以使用更低的算法复杂度来获得近似解。

准备

MiniBatch Kmeans 是 KMeans 的更快实现。KMeans 的计算量非常大，问题是 NPH 的。

但是，使用 MiniBatch KMeans，我们可以将 KMeans 加速几个数量级。这通过处理多个子样本来完成，它们叫做 MiniBatch。如果子样本是收敛的，并且拥有良好的初始条件，就得到了常规 KMeans 的近似解。

操作步骤

让我们对 MiniBatch 聚类做一个概要的性能分析。首先，我们观察总体的速度差异，之后我们会观察估计中的误差。

```
>>> from sklearn.datasets import make_blobs
>>> blobs, labels = make_blobs(int(1e6), 3)

>>> from sklearn.cluster import KMeans, MiniBatchKMeans

>>> kmeans = KMeans(n_clusters=3) >>> minibatch = MiniBatchKMean
s(n_clusters=3)
```

> 要理解这些度量的目的是暴露问题。所以，需要多加小心，来确保跑分的高精度性。这个话题还有大量可用的信息。如果你真的希望了解，MiniBatch KMeans 为何在粒度上更具优势，最好还是要阅读它们。

既然准备已经完成，我们可以测量时间差异：

```
>>> %time kmeans.fit(blobs) #IPython Magic CPU times: user 8.17
s, sys: 881 ms, total: 9.05 s Wall time: 9.97 s

>>> %time minibatch.fit(blobs) CPU times: user 4.04 s, sys: 90.1
 ms, total: 4.13 s Wall time: 4.69 s
```

CPU 时间上有很大差异。聚类性能上的差异在下面展示：

```
>>> kmeans.cluster_centers_[0] array([ 1.10522173, -5.59610761,
-8.35565134])

>>> minibatch.cluster_centers_[0] array([ 1.12071187, -5.61215116
, -8.32015587])
```

我们可能要问的下一个问题就是，两个形心距离多远。

```
>>> from sklearn.metrics import pairwise
>>> pairwise.pairwise_distances(kmeans.cluster_centers_[0],
                                 minibatch.cluster_centers_[0])
array([[ 0.03305309]])
```

看起来十分接近了。对角线包含形心的差异：

```
>>> np.diag(pairwise.pairwise_distances(kmeans.cluster_centers_,
             minibatch.cluster_centers_))
array([ 0.04191979, 0.03133651,  0.04342707])
```

工作原理

这里的批次就是关键。批次被迭代来寻找批次均值。对于下一次迭代来说，前一个批次的均值根据当前迭代来更新。有多种选项，用于控制 KMeans 的通用行为，和决定 MiniBatch KMeans 的参数。

`batch_size` 参数决定批次应为多大。只是玩玩的话，我们可以运行 MiniBatch，但是，此时我们将批次数量设置为和数据集大小相同。

```
>>> minibatch = MiniBatchKMeans(batch_size=len(blobs))
>>> %time minibatch.fit(blobs)
CPU times: user 34.6 s, sys: 3.17 s, total: 37.8 s Wall time: 44
.6 s
```

显然，这就违背了问题的核心，但是这的确展示了重要东西。选择差劲的初始条件可能影响我们的模型，特别是聚类模型的收敛。使用 MiniBatch KMeans，全局最优是否能达到，是不一定的。

3.5 使用 KMeans 聚类来量化图像

图像处理是个重要的话题，其中聚类有一些应用。值得指出的是，Python 中有几种非常不错的图像处理库。Scikit-image 是 Scikit-learn 的“姐妹”项目。如果你打算做任何复杂的事情，都值得看一看它。

准备

我们在这篇秘籍中会有一些乐趣。目标是使用聚类来把图像变模糊。

首先，我们要利用 SciPy 来读取图像。图像翻译为三维数组， `x` 和 `y` 坐标描述了高度和宽度，第三个维度表示每个图像的 RGB 值。

```
# in your terminal
$ wget http://blog.trenthauck.com/assets/headshot.jpg
```

操作步骤

现在，让我们在 Python 中读取图像：

```
>>> from scipy import ndimage
>>> img = ndimage.imread("headshot.jpg")
>>> plt.imshow(img)
```

下面就是图像：



嘿，这就是（年轻时期的）作者。

既然我们已经有了图像，让我们检查它的维度：

```
>>> img.shape
(420, 420, 3)
```

为了实际量化图像，我们需要将其转换为二维数组，长为 `420x420` ，宽为 RGB 值。思考它的更好的方法，是拥有一堆三维空间中的数据点，并且对点进行聚类来降低图像中的不同颜色的数量 -- 这是一个简单的量化方式。

首先，让我们使数组变形，它是个 NumPy 数组，所以非常简单：

```
>>> x, y, z = img.shape
>>> long_img = img.reshape(x*y, z)
>>> long_img.shape (176400, 3)
```

现在我们开始聚类过程。首先，让我们导入聚类模块，并创建 KMeans 对象。我们传入 `n_clusters=5` ，使我们拥有 5 个簇，或者实际上是 5 个不同颜色。

这是个不错的秘籍，我们使用前面提到的轮廓距离：

```
>>> from sklearn import cluster
>>> k_means = cluster.KMeans(n_clusters=5)
>>> k_means.fit(long_img)
```

既然我们已经训练了 KMeans 对象，让我们看看我们的眼色：

```
>>> centers = k_means.cluster_centers_
>>> centers
array([[ 142.58775848, 206.12712986,  226.04416873],
       [  86.29356543,  68.86312505,   54.04770507],
       [ 194.36182899,  172.19845258,  149.65603813],
       [  24.67768412,   20.45778933,   16.19698314],
       [ 149.27801776,  132.19850659,  115.32729167]])
```

工作原理

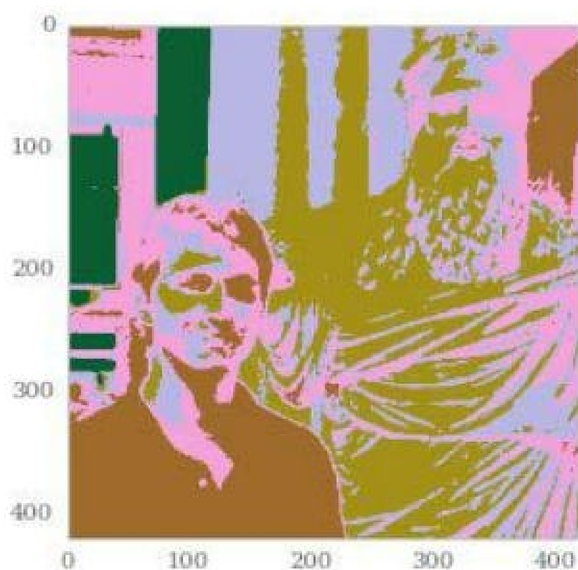
既然我们拥有了形心，我们需要的下一个东西就是标签。它会告诉我们，哪个点关联哪个簇。

```
>>> labels = k_means.labels_
>>> labels[:5] array([1, 1, 1, 1, 1], dtype=int32)
```

这个时候，我们需要最简的 NumPy 操作，之后是一个变形，我们就拥有的新的图像：

```
>>> plt.imshow(centers[labels].reshape(x, y, z))
```

下面就是产生的图像：



3.6 寻找特征空间中的最接近对象

有时，最简单的事情就是求出两个对象之间的距离。我们刚好需要寻找一些距离的度量，计算成对（Pairwise）距离，并将结果与我们的预期比较。

准备

Scikit-learn 中，有个叫做 `sklearn.metrics.pairwise` 的底层工具。它包含一些服务函数，计算矩阵 `X` 中向量之间的距离，或者 `X` 和 `Y` 中的向量距离。

这对于信息检索来说很实用。例如，提供一组客户信息，带有属性 `X` ，我们可能希望选取有个客户代表，并找到与这个客户最接近的客户。实际上，我们可能希望将客户按照相似性度量的概念，使用距离函数来排序。相似性的质量取决于特征空间选取，以及我们在空间上所做的任何变换。

操作步骤

我们会使用 `pairwise_distances` 函数来判断对象的接近程度。要记住，接近程度就像我们用于聚类/分类的距离函数。

首先，让我们从 `metric` 模块导入 `pairwise_distances` 函数，并创建用于操作的数据集：

```
>>> from sklearn.metrics import pairwise
>>> from sklearn.datasets import make_blobs
>>> points, labels = make_blobs()
```

用于检查距离的最简单方式是 `pairwise_distances` ：

```
>>> distances = pairwise.pairwise_distances(points)
```

`distances` 是个 `NxN` 的矩阵，对角线为 0。在最简单的情况中，让我们先看看每个点到第一个点的距离：

```
>>> np.diag(distances) [:5]
array([ 0.,  0.,  0.,  0.,  0.])
```

现在我们可以查找最接近于第一个点的点：

```
>>> distances[0][:5]
array([  0., 11.82643041,1.23751545, 1.17612135, 14.61927874])
```

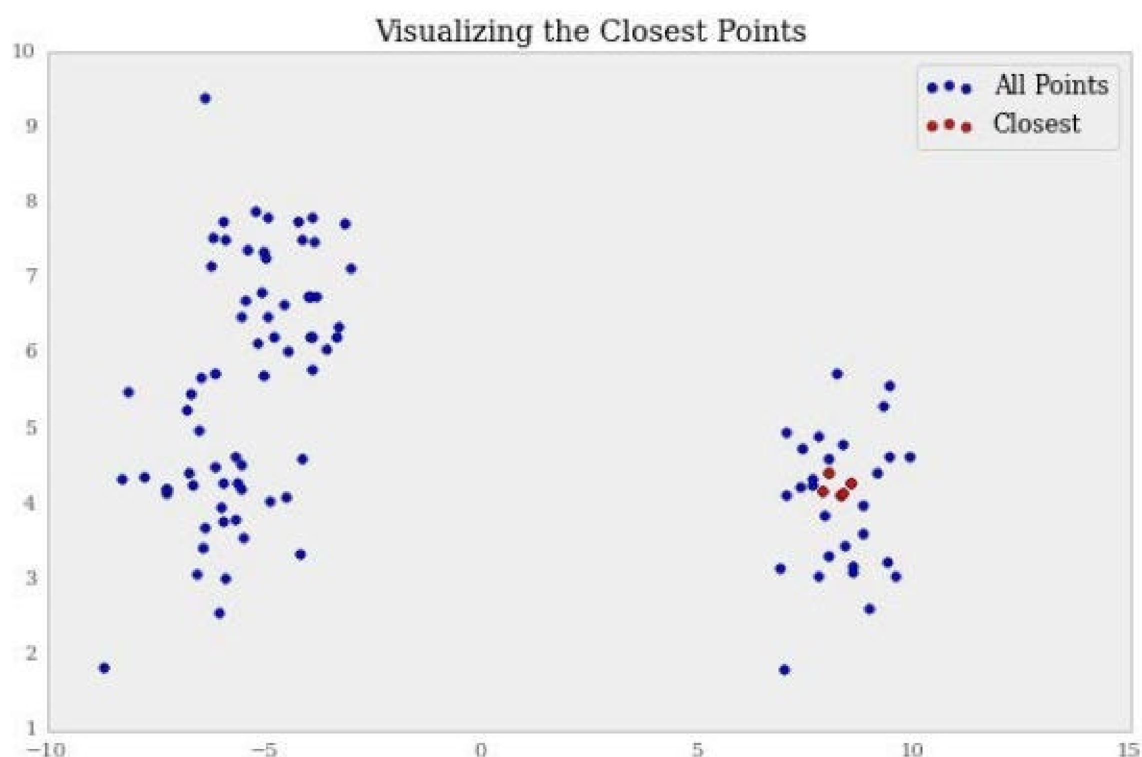
将点按照接近程度排序，很容易使用 `np.argsort` 做到：

```
>>> ranks = np.argsort(distances[0])
>>> ranks[:5]
array([ 0, 27, 98, 23, 67])
```

`argsort` 的好处是，现在我们可以排序我们的 `points` 矩阵，来获得真实的点。

```
>>> points[ranks][:5]
array([[ 8.96147382, -1.90405304],
       [ 8.75417014, -1.76289919],
       [ 8.78902665, -2.27859923],
       [ 8.59694131, -2.10057667],
       [ 8.70949958, -2.30040991]])
```

观察接近的点是什么样子，可能十分有用。结果在意料之中：



工作原理

给定一些距离函数，每个点都以成对函数来度量。通常为欧几里得距离，它是：

$$d(x,y)=\sqrt{\sum_i (x_i-y_i)^2}$$

详细来说，它计算了两个向量每个分量的差，计算它们的平方，求和，之后计算它的平方根。这看起来很熟悉，因为在计算均方误差的时候，我们使用的东西很相似。如果我们计算了平方根，就一样了。实际上，经常使用的度量是均方根误差（RMSE），它就是距离函数的应用。

在 Python 中，这看起来是：

```
>>> def euclid_distances(x, y):
        return np.power(np.power(x - y, 2).sum(), .5)
>>> euclid_distances(points[0], points[1])
11.826430406213145
```

Scikit-learn 中存在一些其他函数，但是 Scikit-learn 也会使用 SciPy 的距离函数。在本书编写之时，Scikit-learn 距离函数支持稀疏矩阵。距离函数的更多信息请查看 SciPy 文档。

- `cityblock`
- `cosine`
- `euclidean`
- `l1`
- `l2`
- `manhattan`

我们现在可以解决问题了。例如，如果我们站在原点处的格子上，并且线是街道，为了到达点 `(5,5)` ，我们需要走多远呢？

```
>>> pairwise.pairwise_distances([[0, 0], [5, 5]], metric='cityblock')[0]
array([  0.,  10.])
```

更多

使用成对距离，我们可以发现位向量之间的相似性。这是汉明距离的事情，它定义为：

$$\sum_i I_{x_i \neq y_i}$$

使用下列命令：

```
>>> X = np.random.binomial(1, .5, size=(2, 4)).astype(np.bool)
>>> X
array([[False,  True, False, False],
       [False, False, False,  True]], dtype=bool)
>>> pairwise.pairwise_distances(X, metric='hamming')
array([[ 0.  ,  0.25],
       [ 0.25,  0.  ]])
```

3.7 使用高斯混合模型的概率聚类

在 KMeans 中，我们假设簇的方差是相等的。这会导致空间的细分，这决定了簇如何被分配。但是，如果有一种场景，其中方差不是相等的，并且每个簇中的点拥有一个与之相关的概率，会怎么样？

准备

有一种更加概率化的方式，用于查看 KMeans 聚类。KMeans 聚类相当于将协方差矩阵 `S` 应用于高斯混合模型，这个矩阵可以分解为单位矩阵成误差。对于每个簇，协方差结构是相同的。这就产生了球形聚类。

但是，如果我们允许 `S` 变化，就可以估计 GMM，并将其用于预测。我们会以单变量的角度看到它的原理，之后扩展为多个维度。

操作步骤

首先，我们需要创建一些数据。例如，让我们模拟女性和男性的身高。我们会在整个秘籍中使用这个例子。这是个简单的例子，但是会展示出我们在 N 维空间中想要完成的东西，这比较易于可视化：

```
>>> import numpy as np
>>> N = 1000

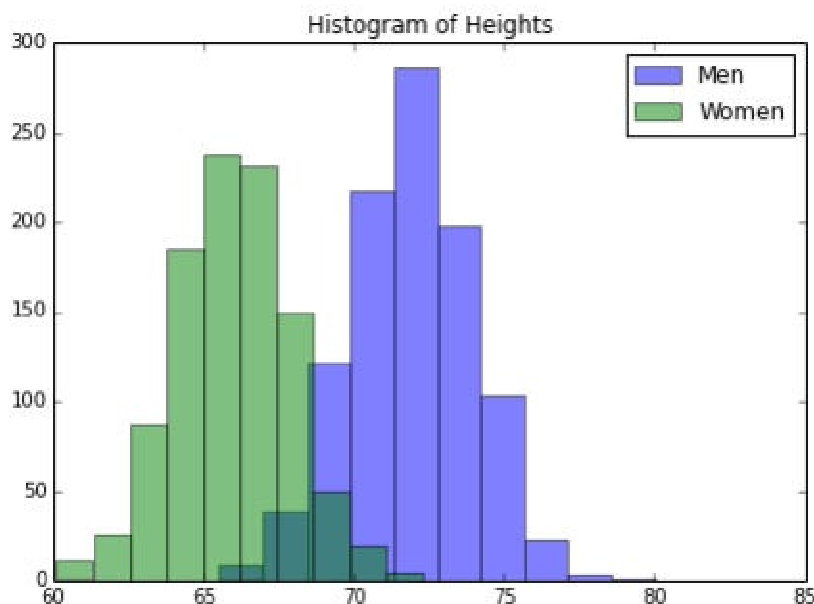
>>> in_m = 72
>>> in_w = 66

>>> s_m = 2
>>> s_w = s_m

>>> m = np.random.normal(in_m, s_m, N)
>>> w = np.random.normal(in_w, s_w, N)
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Histogram of Heights")
>>> ax.hist(m, alpha=.5, label="Men");
>>> ax.hist(w, alpha=.5, label="Women");
>>> ax.legend()
```

下面是输出：



下面，我们的兴趣是，对分组二次抽样，训练分布，之后预测剩余分组。

```
>>> random_sample = np.random.choice([True, False], size=m.size)

>>> m_test = m[random_sample]
>>> m_train = m[~random_sample]

>>> w_test = w[random_sample]
>>> w_train = w[~random_sample]
```

现在我们需要获得男性和女性高度的经验分布，基于训练集：

```
>>> from scipy import stats
>>> m_pdf = stats.norm(m_train.mean(), m_train.std())
>>> w_pdf = stats.norm(w_train.mean(), w_train.std())
```

对于测试集，我们要计算，基于数据点从每个分布中生成的概率，并且最可能的分布会分配合适的标签。当然，我们会看到有多么准确。

```
>>> m_pdf.pdf(m[0])
0.043532673457165431
>>> w_pdf.pdf(m[0])
9.2341848872766183e-07
```

要注意概率中的差异。

假设当男性的概率更高时，我们会猜测，但是如果女性的概率更高，我们会覆盖它。

```
>>> guesses_m = np.ones_like(m_test)
>>> guesses_m[m_pdf.pdf(m_test) < w_pdf.pdf(m_test)] = 0
```

显然，问题就是我们有多么准确。由于正确情况下 `guesses_m` 为 1，否则为 0，我们计算向量的均值来获取准确度。

```
>>> guesses_m.mean()
0.93775100401606426
```

不是太糟。现在，来看看我们在女性的分组中做的有多好，使用下面的命令：

```
>>> guesses_w = np.ones_like(w_test)
>>> guesses_w[m_pdf.pdf(w_test) > w_pdf.pdf(w_test)] = 0
>>> guesses_w.mean() 0.93172690763052213
```

让我们允许两组间的方差不同。首先，创建一些新的数组：

```
>>> s_m = 1
>>> s_w = 4

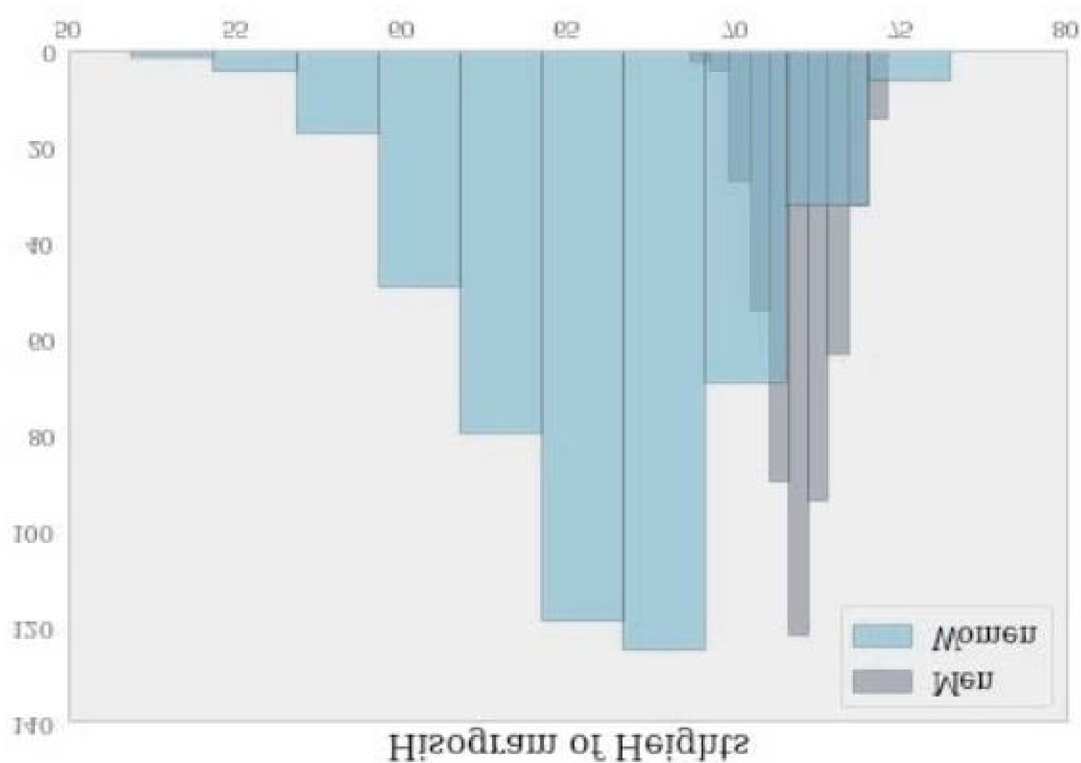
>>> m = np.random.normal(in_m, s_m, N)
>>> w = np.random.normal(in_w, s_w, N)
```

之后，创建训练集：

```
>>> m_test = m[random_sample]
>>> m_train = m[~random_sample]

>>> w_test = w[random_sample]
>>> w_train = w[~random_sample]
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Histogram of Heights")
>>> ax.hist(m_train, alpha=.5, label="Men");
>>> ax.hist(w_train, alpha=.5, label="Women");
>>> ax.legend()
```

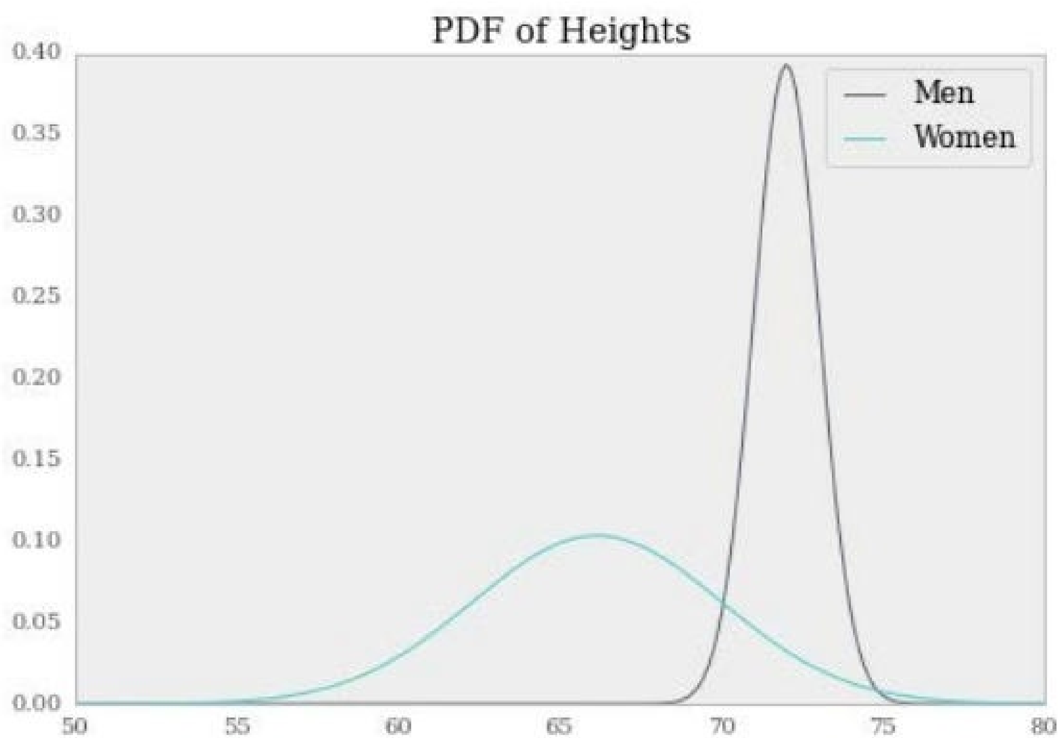
让我们看看男性和女性之间的方差差异：



现在我们可以创建相同的 PDF：

```
>>> m_pdf = stats.norm(m_train.mean(), m_train.std())
>>> w_pdf = stats.norm(w_train.mean(), w_train.std())
```

下面是输出：

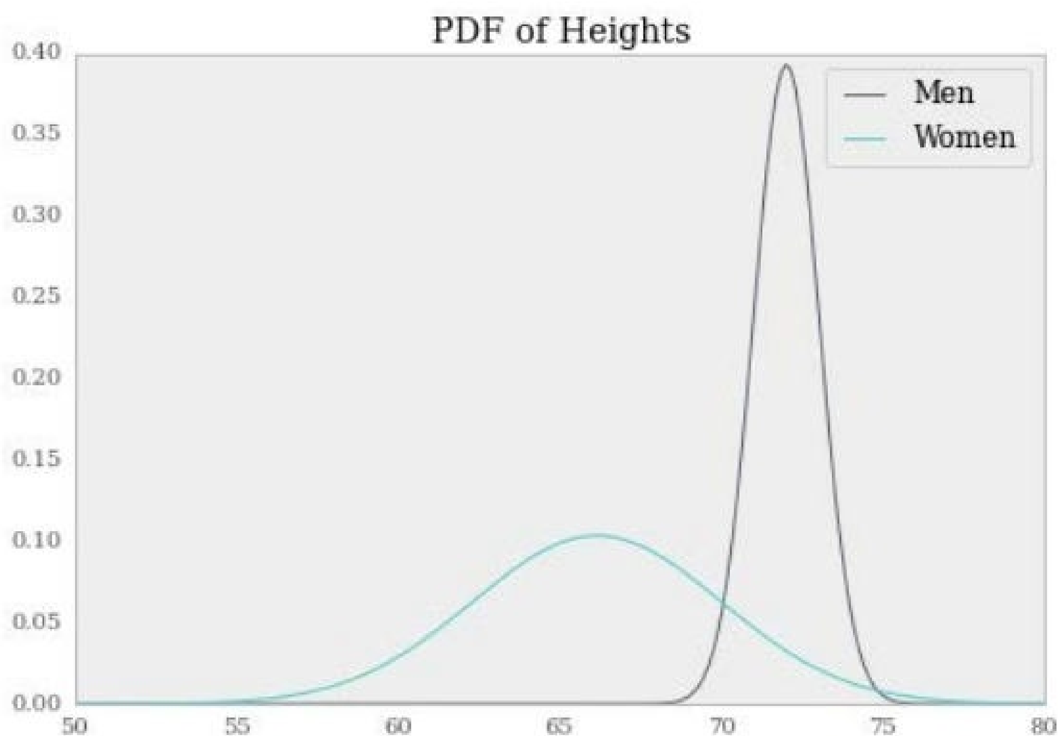


你可以在多维空间中想象他：

```
>>> class_A = np.random.normal(0, 1, size=(100, 2))
>>> class_B = np.random.normal(4, 1.5, size=(100, 2))
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.scatter(class_A[:,0], class_A[:,1], label='A', c='r')
>>> ax.scatter(class_B[:,0], class_B[:,1], label='B')
```

下面是输出：



工作原理

好的，所以既然我们看过了，我们基于分布对点分类的方式，让我们看看如何在 Scikit 中首先：

```
>>> from sklearn.mixture import GMM
>>> gmm = GMM(n_components=2)
>>> X = np.row_stack((class_A, class_B))
>>> y = np.hstack((np.ones(100), np.zeros(100)))
```

由于我们是小巧的数据科学家，我们创建训练集：

```
>>> train = np.random.choice([True, False], 200)
>>> gmm.fit(X[train]) GMM(covariance_type='diag', init_params='w
mc', min_covar=0.001,
     n_components=2, n_init=1, n_iter=100, params='wmc',
     random_state=None,    thresh=0.01)
```

训练和预测的完成方式，和 Scikit-learn 的其它对象相同。

```
>>> gmm.fit(X[train])
>>> gmm.predict(X[train])[:5]
array([0, 0, 0, 0, 0])
```

既然模型已经训练了，有一些值得一看的其它方法。

例如，使用 `score_examples` ，我们实际上可以为每个标签获得每个样例的可能性。

3.8 将 KMeans 用于离群点检测

这一章中，我们会查看 Kmeans 离群点检测的机制和正义。它对于隔离一些类型的错误很实用，但是使用时应多加小心。

准备

这个秘籍中，我们会使用 KMeans，对簇中的点执行离群点检测。要注意，提及离群点和离群点检测时有很多“阵营”。以便面，我们可能通过移除离群点，来移除由数据生成过程生成的点。另一方面，离群点可能来源于测量误差或一些其它外部因素。

这就是争议的重点。这篇秘籍的剩余部分有关于寻找离群点。我们的假设是，我们移除离群点的选择是合理的。

离群点检测的操作是，查找簇的形心，之后通过点到形心的距离来识别潜在的离群点。

操作步骤

首先，我们会生成 100 个点的单个数据块，之后我们会识别 5 个离形心最远的点。它们就是潜在的离群点。

```
>>> from sklearn.datasets import make_blobs
>>> X, labels = make_blobs(100, centers=1)
>>> import numpy as np
```

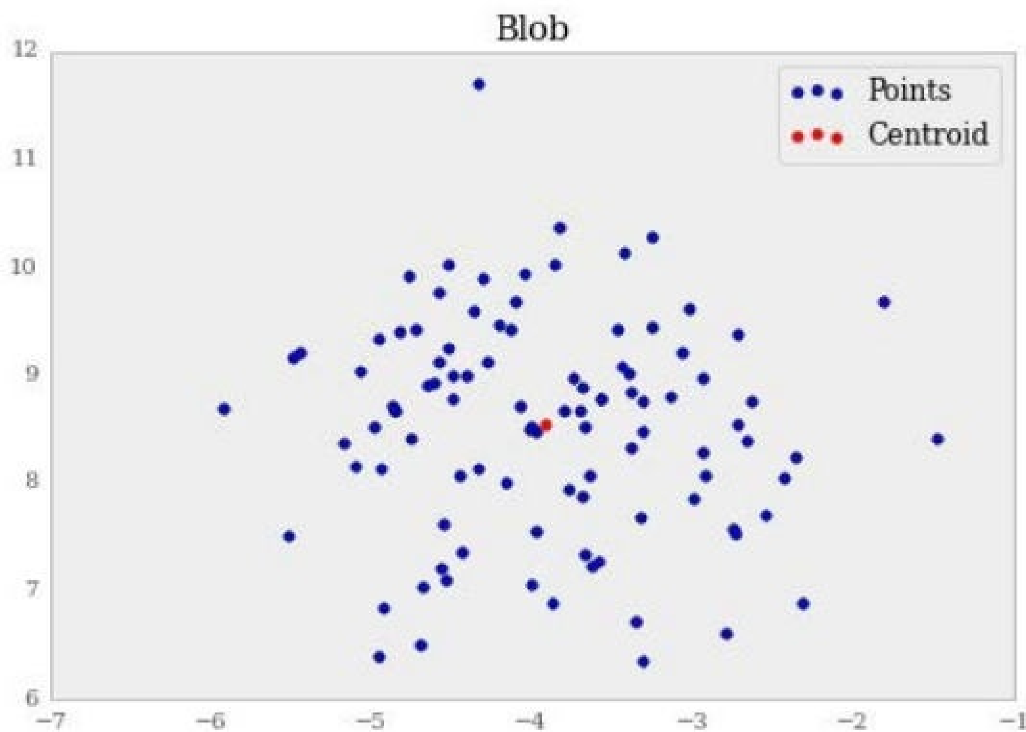
非常重要的是，Kmeans 聚类只有一个形心。这个想法类似于用于离群点检测的单类 SVM。

```
>>> from sklearn.cluster import KMeans
>>> kmeans = KMeans(n_clusters=1)
>>> kmeans.fit(X)
```

现在，让我们观察绘图。对于那些远离中心的点，尝试猜测哪个点会识别为五个离群点之一：

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Blob")
>>> ax.scatter(X[:, 0], X[:, 1], label='Points')
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
                kmeans.cluster_centers_[:, 1],
                label='Centroid',
                color='r')
>>> ax.legend()
```

下面就是输出：



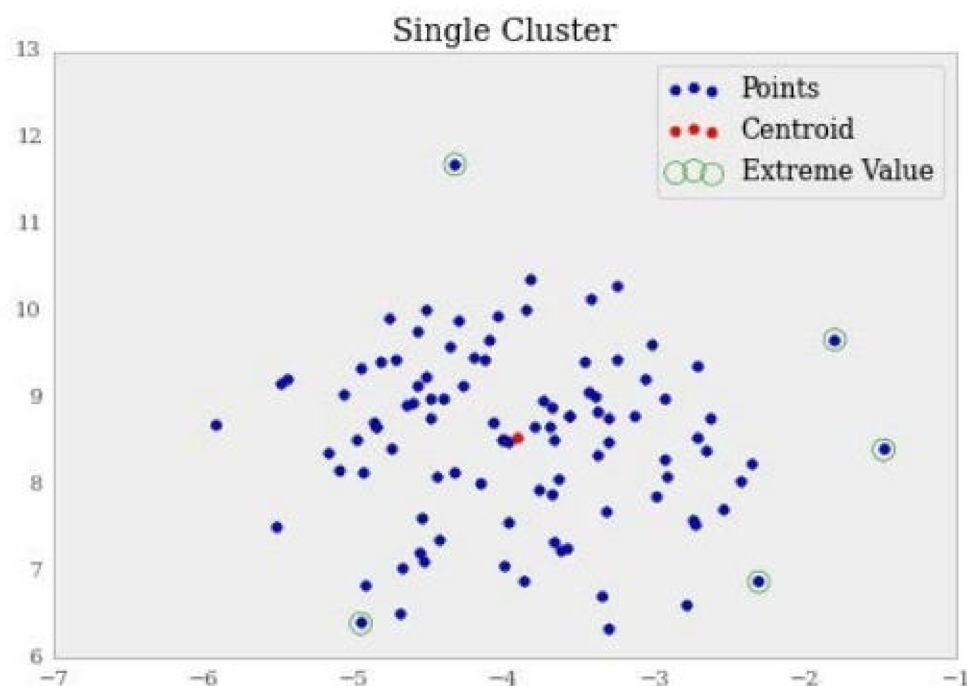
现在，让我们识别五个最接近的点：

```
>>> distances = kmeans.transform(X)
# argsort returns an array of indexes which will sort the array
in ascending order
# so we reverse it via [::-1] and take the top five with [:5]
>>> sorted_idx = np.argsort(distances.ravel())[::-1][:5]
```

现在，让我们看看哪个点离得最远：

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Single Cluster")
>>> ax.scatter(X[:, 0], X[:, 1], label='Points')
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
                kmeans.cluster_centers_[:, 1],
                label='Centroid', color='r')
>>> ax.scatter(X[sorted_idx][:, 0], X[sorted_idx][:, 1],
                label='Extreme Value', edgecolors='g',
                facecolors='none', s=100)
>>> ax.legend(loc='best')
```

下面是输出：



如果我们喜欢的话，移除这些点很容易。

```
>>> new_X = np.delete(X, sorted_idx, axis=0)
```

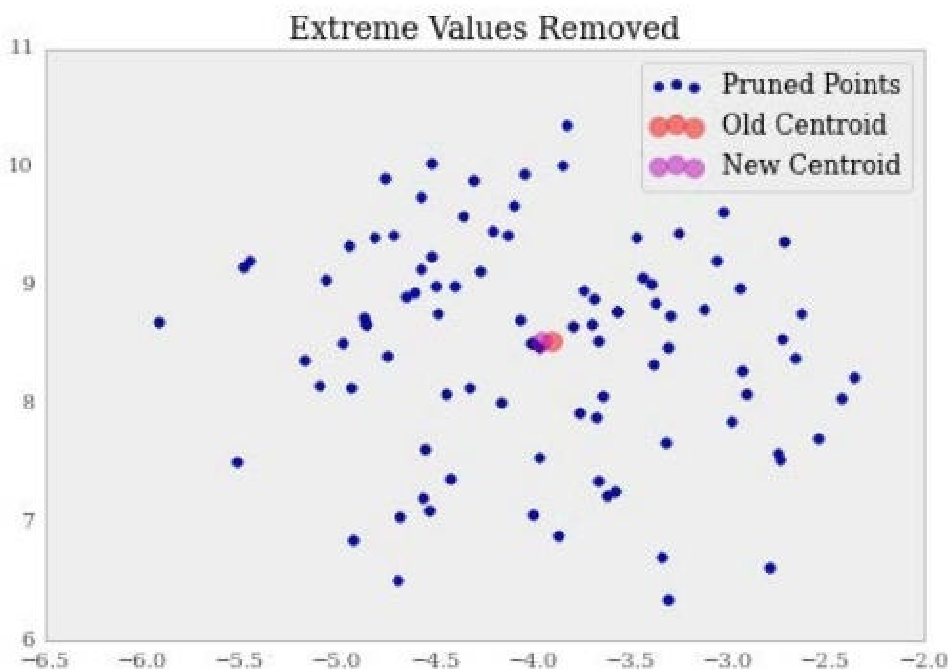
同样，移除这些点之后，形心明显变化了。

```
>>> new_kmeans = KMeans(n_clusters=1)
>>> new_kmeans.fit(new_X)
```

让我们将旧的和新的形心可视化：

```
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.set_title("Extreme Values Removed")
>>> ax.scatter(new_X[:, 0], new_X[:, 1], label='Pruned Points')
>>> ax.scatter(kmeans.cluster_centers_[:, 0],
               kmeans.cluster_centers_[:, 1], label='Old Centroi
d',
               color='r', s=80, alpha=.5)
>>> ax.scatter(new_kmeans.cluster_centers_[:, 0],
               new_kmeans.cluster_centers_[:, 1], label='New Cen
troid',
               color='m', s=80, alpha=.5)
>>> ax.legend(loc='best')
```

下面是输出：



显然，形心没有移动多少，仅仅移除五个极端点时，我们的预期就是这样。这个过程可以重复，知道我们对数据表示满意。

工作原理

我们已经看到，高斯分布和 KMeans 聚类之间有本质联系。让我们基于形心和样本的协方差矩阵创建一个经验高斯分布，并且查看每个点的概率 -- 理论上是我们溢出的五个点。这刚好展示了，我们实际上溢出了拥有最低可能性的值。距离和可能性之间的概念十分重要，并且在你的机器学习训练中会经常出现。

使用下列命令来创建经验高斯分布：

```
>>> from scipy import stats
>>> emp_dist = stats.multivariate_normal(
               kmeans.cluster_centers_.ravel())
>>> lowest_prob_idx = np.argsort(emp_dist.pdf(X))[:5]
>>> np.all(X[sorted_idx] == X[lowest_prob_idx]) True
```

3.9 将 KNN 用于回归

回归在这本书的其它地方有所设计，但是我们可能打算在特征空间的“口袋”中运行回归。我们可以认为，我们的数据集要经过多道数据处理工序。如果是这样，只训练相似数据点是个不错的想法。

准备

我们的老朋友，回归，可以用于聚类的上下文中。回归显然是个监督学习技巧，所以我们使用 KNN 而不是 KMeans。

对于 KNN 回归来说，我们使用特征空间中的 K 个最近点，来构建回归，而不像常规回归那样使用整个特征空间。

操作步骤

对于这个秘籍，我们使用 `iris` 数据集。如果我们打算预测一些东西，例如每朵花的花瓣宽度，根据 `iris` 物种来聚类可能会给我们更好的结果。KNN 回归不会根据物种来聚类，但我们的假设是，相同物种的 X 会接近，或者这个案例中，是花瓣长度。

对于这个秘籍，我们使用 `iris` 数据集：

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> iris.feature_names ['sepal length (cm)', 'sepal width (cm)',
'petal length (cm)',  'petal width (cm)']
```

我们尝试基于萼片长度和宽度来预测花瓣长度。我们同时训练一个线性回归，来对比观察 KNN 回归有多好。

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression()
>>> lr.fit(X, y)
>>> print "The MSE is: {:.2}".format(np.power(y - lr.predict(X),
           2).mean())
The MSE is: 0.15
```

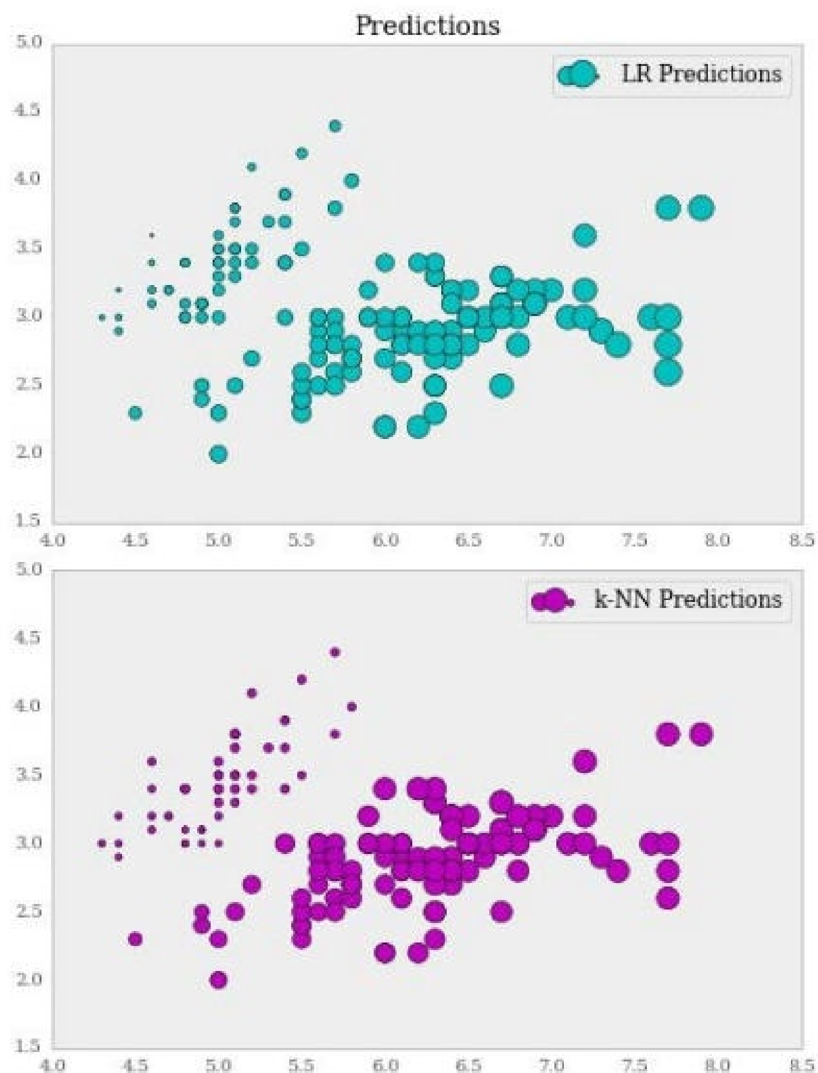
现在，对于 KNN 回归，使用下列代码：

```
>>> from sklearn.neighbors import KNeighborsRegressor
>>> knnr = KNeighborsRegressor(n_neighbors=10)
>>> knnr.fit(X, y)
>>> print "The MSE is: {:.2}".format(np.power(y - knnr.predict(X
),
           2).mean())
The MSE is: 0.069
```

让我们看看，当我们让它使用最接近的 10 个点用于回归时，KNN 回归会做什么？

```
>>> f, ax = plt.subplots(nrows=2, figsize=(7, 10))
>>> ax[0].set_title("Predictions")
>>> ax[0].scatter(X[:, 0], X[:, 1], s=lr.predict(X)*80, label='L
R
    Predictions', color='c', edgecolors='black')
>>> ax[1].scatter(X[:, 0], X[:, 1], s=knnr.predict(X)*80, label=
'k-NN
    Predictions', color='m', edgecolors='black')
>>> ax[0].legend()
>>> ax[1].legend()
```

输出如下：



很显然，预测大部分都是接近的。但是让我们与实际情况相比，看看 Setosa 物种的预测：

```
>>> setosa_idx = np.where(iris.target_names=='setosa')
>>> setosa_mask = iris.target == setosa_idx[0]
>>> y[setosa_mask][:5] array([ 0.2,  0.2,  0.2,  0.2,  0.2])
>>> knnr.predict(X)[setosa_mask][:5]
array([ 0.28,  0.17,  0.21,  0.2 ,  0.31])
>>> lr.predict(X)[setosa_mask][:5]
array([ 0.44636645, 0.53893889, 0.29846368, 0.27338255, 0.326128
85])
```

再次观察绘图，Setosa 物种（左上方的簇）被线性回归估计过高，但是 KNN 非常接近真实值。

工作原理

KNN 回归非常简单，它计算被测试点的 K 个最接近点的均值。

让我们手动预测单个点：

```
>>> example_point = X[0
```

现在，我们需要获取离我们的 `our_example_point` 最近的 10 个点：

```
>>> from sklearn.metrics import pairwise
>>> distances_to_example = pairwise.pairwise_distances(X)[0]
>>> ten_closest_points = X[np.argsort(distances_to_example)][:10
]
>>> ten_closest_y = y[np.argsort(distances_to_example)][:10]
>>> ten_closest_y.mean()
0.28000
```

我们可以看到它非常接近预期。

第四章 使用 scikit-learn 对数据分类

> 作者：Trent Hauck
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

分类在大量语境下都非常重要。例如，如果我们打算自动化一些决策过程，我们可以利用分类。在我们需要研究诈骗的情况下，有大量的事务，人去检查它们是不实际的。所以，我们可以使用分类都自动化这种决策。

4.1 使用决策树实现基本的分类

这个秘籍中，我们使用决策树执行基本的分类。它们是非常不错的模型，因为它们很易于理解，并且一旦训练完成，评估就很容易。通常可以使用 SQL 语句，这意味着结果可以由许多人使用。

准备

这个秘籍中，我们会看一看决策树。我喜欢将决策树看做基类，大量的模型从中派生。它是个非常简单的想法，但是适用于大量的情况。

首先，让我们获取一些分类数据，我们可以使用它来练习：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=1000, n_featur
es=3,
                                        n_redundant=0)
```

操作步骤

处理决策树非常简单。我们首先需要导入对象，之后训练模型：

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y) DecisionTreeClassifier(compute_importances=None
, criterion='gini',
                        max_depth=None, max_features=None,
                        max_leaf_nodes=None, min_density=None,
                        min_samples_leaf=1, min_samples_split=2,
                        random_state=None, splitter='best')
>>> preds = dt.predict(X)
>>> (y == preds).mean()
1.0
```

你可以看到，我们猜测它是正确的。显然，这只是凑合着运行。现在我们研究一些选项。

首先，如果你观察 `dt` 对象，它拥有多种关键字参数，决定了对象的行为。我们如何选择对象十分重要，所以我们要详细观察对象的效果。

我们要观察的第一个细节是 `max_depth` 。这是个重要的参数，决定了允许多少分支。这非常重要，因为决策树需要很长时间来生成样本外的数据，它们带有一些类型的正则化。之后，我们会看到，我们如何使用多种浅层决策树，来生成更好的模型。让我们创建更复杂的数据集并观察当我们允许不同 `max_depth` 时会发生什么。我们会将这个数据及用于剩下的秘籍。

```
>>> n_features=200
>>> X, y = datasets.make_classification(750, n_features,
                                        n_informative=5)
>>> import numpy as np
>>> training = np.random.choice([True, False], p=[.75, .25],
                                size=len(y))

>>> accuracies = []

>>> for x in np.arange(1, n_features+1):
>>> dt = DecisionTreeClassifier(max_depth=x)
>>> dt.fit(X[training], y[training])
>>> preds = dt.predict(X[~training])
>>> accuracies.append((preds == y[~training]).mean())

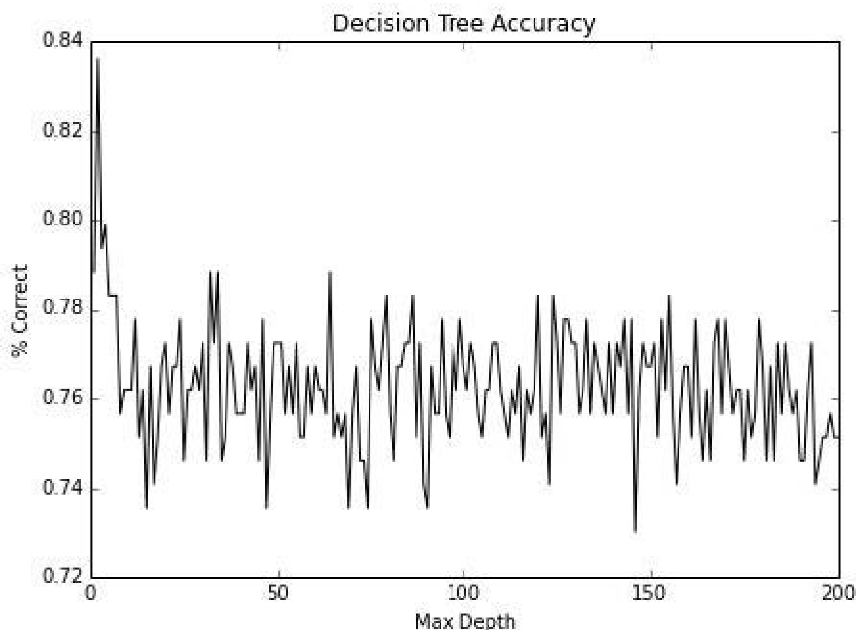
>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.plot(range(1, n_features+1), accuracies, color='k')

>>> ax.set_title("Decision Tree Accuracy")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("Max Depth")
```

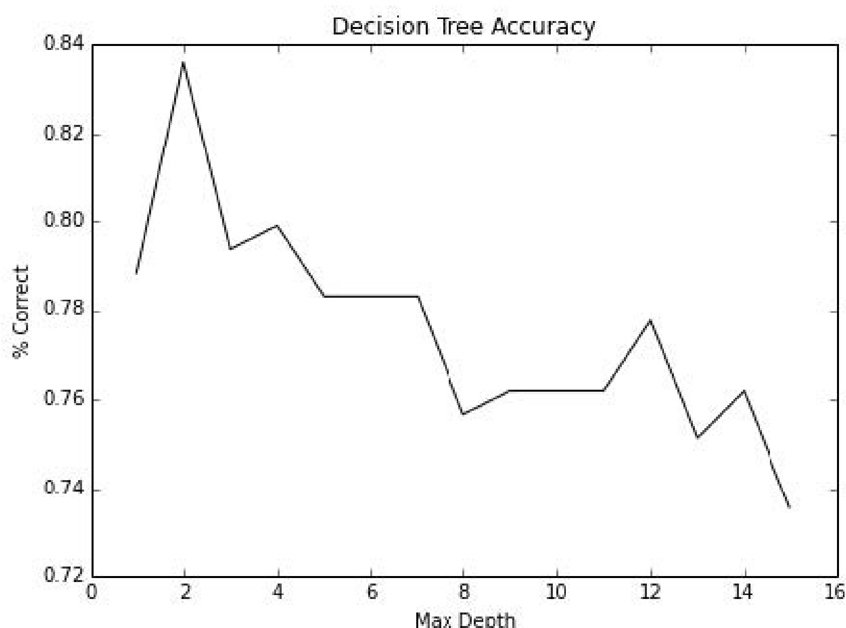
输出如下：



我们可以看到，我们实际上在较低最大深度处得到了漂亮的准确率。让我们进一步看看低级别的准确率，首先是 15：

```
>>> N = 15
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.plot(range(1, n_features+1)[:N], accuracies[:N], color='k'
)
>>> ax.set_title("Decision Tree Accuracy")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("Max Depth")
```

输出如下：



这个就是我们之前看到的峰值。比较令人惊讶的是它很快就下降了。最大深度 1 到 3 可能几乎是相等的。决策树很擅长分离规则，但是需要控制。

我们观察 `compute_importances` 参数。对于随机森林来说，它实际上拥有更广泛的含义。但是我们要更好地了解它。同样值得注意的是，如果你使用了 0.16 或之前的版本，你可以尽管这样做：

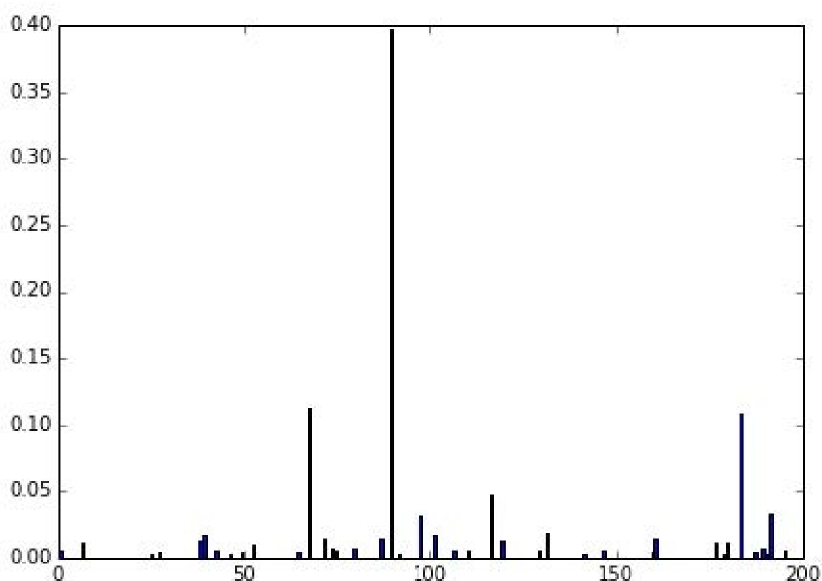
```
>>> dt_ci = DecisionTreeClassifier(compute_importances=True)
>>> dt.fit(X, y)
#plot the importances
>>> ne0 = dt.feature_importances_ != 0

>>> y_comp = dt.feature_importances_[ne0]
>>> x_comp = np.arange(len(dt.feature_importances_))[ne0]

>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.bar(x_comp, y_comp)
```

输出如下：



> 要注意，你可能会得到一个错误，让你知道你不再需要显式设置 `compute_importances` 。

我们看到，这些特征之一非常重要，后面跟着几个其它特征。

工作原理

简单来说，我们所有时间都在构造决策树。当思考场景以及将概率分配给结果时，我们构造了决策树。我们的规则更加复杂，并涉及很多上下文，但是使用决策树，我们关心的所有东西都是结果之间的差异，假设特征的一些信息都是已知的。

现在，让我们讨论熵和基尼系数之间的差异。

熵不仅仅是给定变量的熵值，如果我们知道元素的值，它表示了熵中的变化。这叫做信息增益（IG），数学上是这样：

```
IG(Data,KnownFeatures) = H(Data) - H(Data|KnownFeatures)
```

对于基尼系数，我们关心的是，提供新的信息，一个数据点有多可能被错误标记。

熵和基尼系数都有优缺点。也就是说，如果你观察它们工作方式的主要差异，这可能是个重新验证你的假设的好方式。

4.2 调整决策树模型

如果我们仅仅使用基本的决策树实现，可能拟合得不是很好。所以我们需要调参，以便获得更好的拟合。这非常简单，并且不用花费什么精力。

准备

这个秘籍中，我们深入了解如何调整决策树分类器。有几个渲染，并且在上一个秘籍中，我们仅仅查看了它们之一。

我们会训练一个基本的模型，并实际观察决策树是什么样子。之后，我们会重新检测每个决策，并且指出不同的修改如何影响结构。

如果你打算遵循这个秘籍，需要安装 `pydot` 。

操作步骤

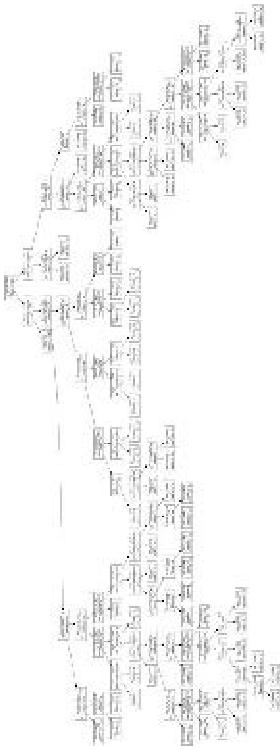
比起其它算法，决策树有许多“把手”，因为我们旋转把手时，易于发现发生了什么。

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(1000, 20, n_informative=3
)
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
```

好的，所以既然我们训练了基本的分类器，我们可以快速地查看它：

```
>>> from StringIO import StringIO
>>> from sklearn import tree
>>> import pydot
>>> str_buffer = StringIO()
>>> tree.export_graphviz(dt, out_file=str_buffer)
>>> graph = pydot.graph_from_dot_data(str_buffer.getvalue())
>>> graph.write("myfile.jpg")
```

这张图几乎难以辨认，但是这展示了一颗复杂的树，它可以使用非最优的决策树，作为结果生成。



哇哦！这是个非常复杂的树，看上去对数据过拟合了。首先，让我们降低最大深度值：

```
>>> dt = DecisionTreeClassifier(max_depth=5)
>>> dt.fit(X, y);
```

顺带一说，如果你想知道为什么能看到分号，通常是 `repr` ，它实际上是决策树的模型。例如 `fit` 函数实际上返回决策树对象，它允许链式调用：

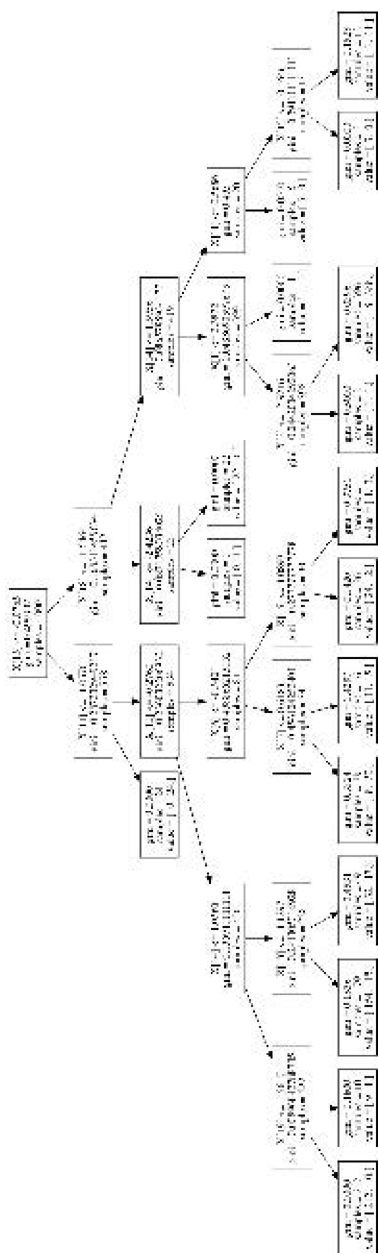
```
>>> dt = DecisionTreeClassifier(max_depth=5).fit(X, y)
```

现在，让我们返回正常的程序。

由于我们会多次绘制它，我们创建一个函数。

```
>>> def plot_dt(model, filename):
       str_buffer = StringIO()
>>> tree.export_graphviz(model, out_file=str_buffer)
>>> graph = pydot.graph_from_dot_data(str_buffer.getvalue())
>>> graph.write_jpg(filename)
>>> plot_dt(dt, "myfile.png")
```

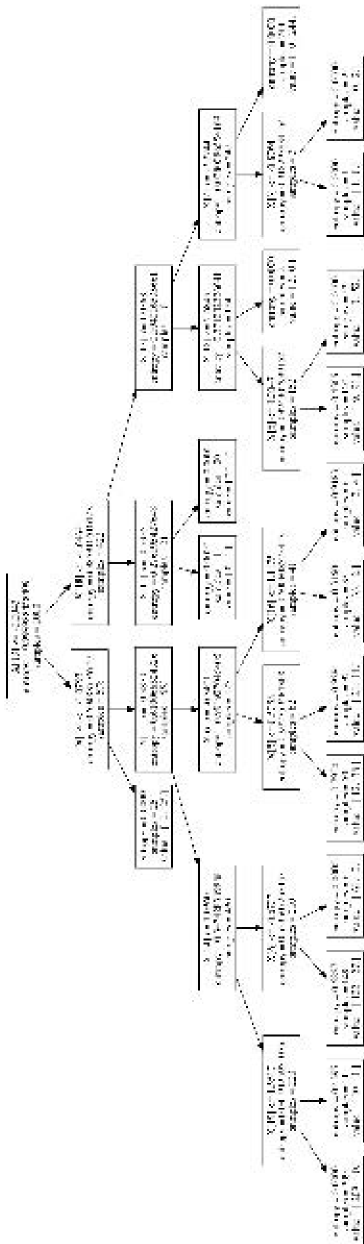
会生成下面的图：



这棵树稍微简单了一些。让我们看看，如果我们将熵用作分割标准，会发生什么：

```
>>> dt = DecisionTreeClassifier(criterion='entropy',
                                 max_depth=5).fit(X, y)
>>> plot(dt, "entropy.png")
```

会生成下面的图：



很容易看到，前两个分割是相同特征，之后的分割以相似总数分布。这是个良好的合理的检查。

同样，注意第一个分割的熵是 0.999，但是使用基尼系数的时候是 0.5。我们需要弄清楚，决策树的分割的两种度量有什么不同。更多信息请见下面的工作原理一节。但是，如果我们想要使用熵穿件决策树，我们必须使用下列命令：

```
>>> dt = DecisionTreeClassifier(min_samples_leaf=10,
                                criterion='entropy',
                                max_depth=5).fit(X, y)
```

工作原理

决策树，通常容易出现过拟合。由于它的自身特性，决策树经常会过拟合，所以，我们需要思考，如何避免过拟合，这是为了避免复杂性。实战中，简单的模型通常会执行得更好。

我们即将在实战中看到这个理念。随机森林会在简单模型的理念上构建。

4.3 使用许多决策树 -- 随机森林

这个秘籍中，我们会将随机森林用于分类任务。由于随机森林对于过拟合非常健壮，并且在大量场景中表现良好，所以使用它。

准备

我们会在工作原理一节中深入探索，但是随即森林通过构造大量浅层树，之后让每颗树为分类投票，再选取投票结果。这个想法在机器学习中十分有效。如果我们发现简单训练的分类器只有 60% 的准确率，我们可以训练大量分类器，它们通常是正确的，并且随后一起使用它们。

操作步骤

训练随机森林分类器的机制在 Scikit 中十分容易。这一节中，我们执行以下步骤：

1. 创建用于练习的样例数据集
2. 训练基本的随机森林对象
3. 看一看训练对象的一些属性

下一个秘籍中，我们会观察如何调整随机森林分类器，让我们以导入数据集来开始：

```
>>> from sklearn import datasets
```

之后，使用 1000 个样例创建数据集：

```
>>> X, y = datasets.make_classification(1000)
```

既然我们拥有了数据，我们可以创建分类器对象并训练它：

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> rf = RandomForestClassifier()
>>> rf.fit(X, y)
```

我们想做的第一件事，就是看看如何拟合训练数据。我们可以将 `predict` 方法用于这些东西：

```
>>> print "Accuracy:\t", (y == rf.predict(X)).mean()
Accuracy:    0.993
>>> print "Total Correct:\t", (y == rf.predict(X)).sum() Total
Correct:    993
```

现在，让我们查看一些属性和犯法。

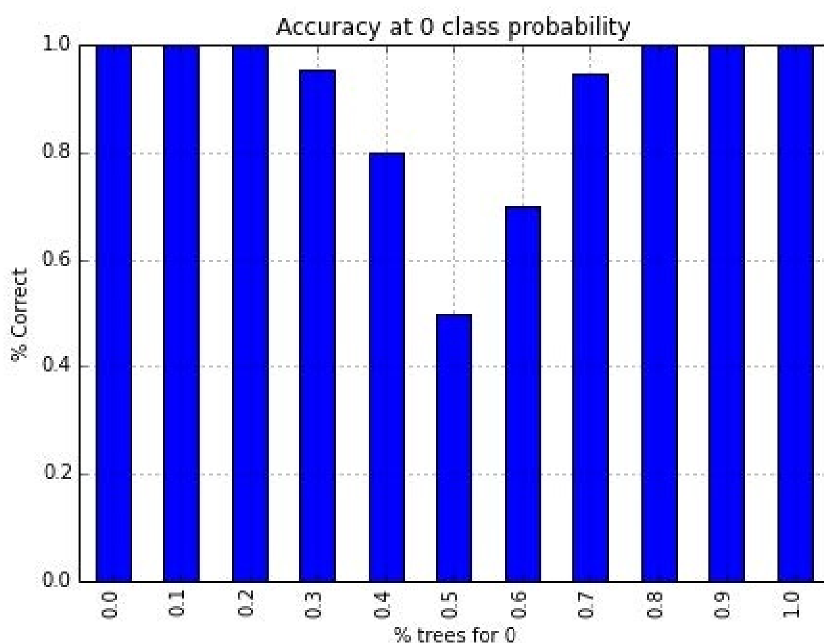
首先，我们查看一些实用属性。这里，由于我们保留默认值，它们是对象的默认值：

- `rf.criterion` ：这是决定分割的标准。默认是 `gini` 。
- `rf.bootstrap` ：布尔值，表示在训练随机森林时是否使用启动样例
- `rf.n_jobs` ：训练和预测的任务数量。如果你打算使用所有处理器，将其设置为 `-1` 。要记住，如果你的数据集不是非常大，使用过多任务通常会导致浪费，因为处理器之间需要序列化和移动。
- `rf.max_features` ：这表示执行最优分割时，考虑的特征数量。在调参过程中这会非常方便。
- `rf.conpute_importtances` ：这有助于我们决定，是否计算特征的重要性。如何使用它的信息，请见更多一节。
- `rf.max_depth` ：这表示树的深度。

有许多属性需要注意，更多信息请查看官方文档。

不仅仅是 `predect` 方法很实用，我们也可以从独立的样子获取概率。这是个非常实用的特性，用于理解每个预测的不确定性。例如，我们可以预测每个样例对于不同类的概率。

```
>>> probs = rf.predict_proba(X)
>>> import pandas as pd
>>> probs_df = pd.DataFrame(probs, columns=['0', '1']) >>> probs
_df['was_correct'] = rf.predict(X) == y
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> probs_df.groupby('0').was_correct.mean().plot(kind='bar', ax
=ax)
>>> ax.set_title("Accuracy at 0 class probability")
>>> ax.set_ylabel("% Correct")
>>> ax.set_xlabel("% trees for 0")
```



工作原理

随机森轮实用预定义数量的弱决策树，并且使用数据的自己训练每一颗树。这对于避免过拟合至关重要。这也是 `bootstrap` 参数的原因。我们的每个树拥有下列东西：

- 票数最多的类
- 输出，如果我们使用回归树

当然，它们是表现上的考量，这会在下一个秘籍中设计。但是出于礼节随机森林如何工作的目的，我们训练一些平均数，作为结果，获得了非常好的分类器。

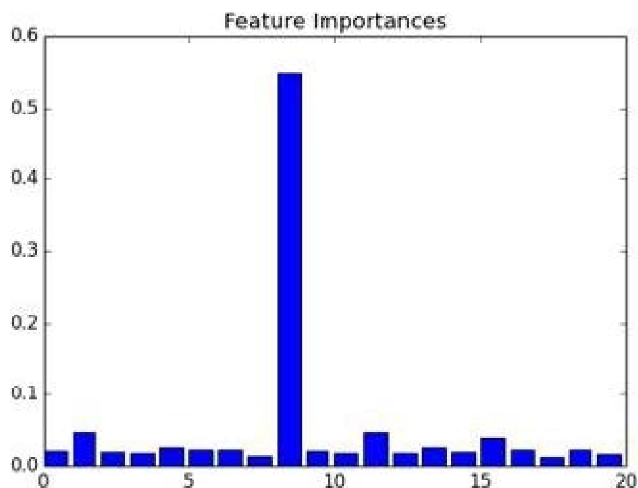
更多

特征重要性是随机森林的不错的副产品。这通常有助于回答一个问题：如果我们拥有 10 个特征，对于判断数据点的真实类别，哪个特征是最重要的？真实世界中的应用都易于观察。例如，如果一个事务是不真实的，我们可能想要了解，是否有特定的信号，可以用于更快弄清楚事务的类别。

如果我们打算极端特征重要性，我们需要在我们创建对象时说明。如果你使用 scikit-learn 0.15，你可能会得到一个警告，说这不是必需的。在 0.16 中，警告会被移除。

```
>>> rf = RandomForestClassifier(compute_importances=True)
>>> rf.fit(X, y)
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> ax.bar(range(len(rf.feature_importances_)),             rf.fe
ature_importances_)
>>> ax.set_title("Feature Importances")
```

下面就是输出：



我们可以看到，在判断结果是类 0 或者 1 时，特定的特征比其它特征重要。

4.4 调整随机森林模型

在上一个秘籍中，我们学习了如何使用随机森林分类器。在这个秘籍中，我们会浏览如何通过调整参数来调整它的表现。

准备

为了调整随机森林模型，我们首先需要创建数据集，它有一些难以预测。之后，我们修改参数并且做一些预处理来更好地拟合数据集。

所以，让我们首先创建数据集：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=10000,
                                        n_features=20,
                                        n_informative=15,
                                        flip_y=.5, weights=[.2,
.8])
```

操作步骤

这个秘籍中，我们执行下列操作：

1. 创建训练和测试集。和上个秘籍不同，如果不与训练集比较，调整模型就毫无用途。
2. 训练基本的随机森林，来评估原始算法有多好。
3. 用系统化的方式修改一些参数，之后观察拟合会发生什么变化。

好了，启动解释器并导入 NumPy：

```
>>> import numpy as np
>>> training = np.random.choice([True, False], p=[.8, .2],
                                size=y.shape)
>>> from sklearn.ensemble import RandomForestClassifier
>>> rf = RandomForestClassifier()
>>> rf.fit(X[training], y[training])
>>> preds = rf.predict(X[~training])
>>> print "Accuracy:\t", (preds == y[~training]).mean()
Accuracy: 0.652239557121
```

我打算用一些花招，引入模型评估度量之一，我们会在这本书后面讨论它。准确率是第一个不错的度量，但是使用混淆矩阵会帮助我们理解发生了什么。

让我们迭代 `max_features` 的推荐选项，并观察对拟合有什么影响。我们同事迭代一些浮点值，它们是所使用的特征的分数。使用下列命令：

```
>>> from sklearn.metrics import confusion_matrix
>>> max_feature_params = ['auto', 'sqrt', 'log2', .01, .5, .99]
>>> confusion_matrixes = {}
>>> for max_feature in max_feature_params:
        rf = RandomForestClassifier(max_features=max_feature)
        rf.fit(X[training], y[training])
>>> confusion_matrixes[max_feature] = confusion_matrix(y[~traini
ng])
>>> rf.predict(X[~training])).ravel()
```

由于我使用了 `ravel` 方法，我们的二维数组现在变成了一维。

现在，导入 Pandas 并查看刚刚创建的混淆矩阵：

```
>>> import pandas as pd

>>> confusion_df = pd.DataFrame(confusion_matrixes)

>>> import itertools
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))

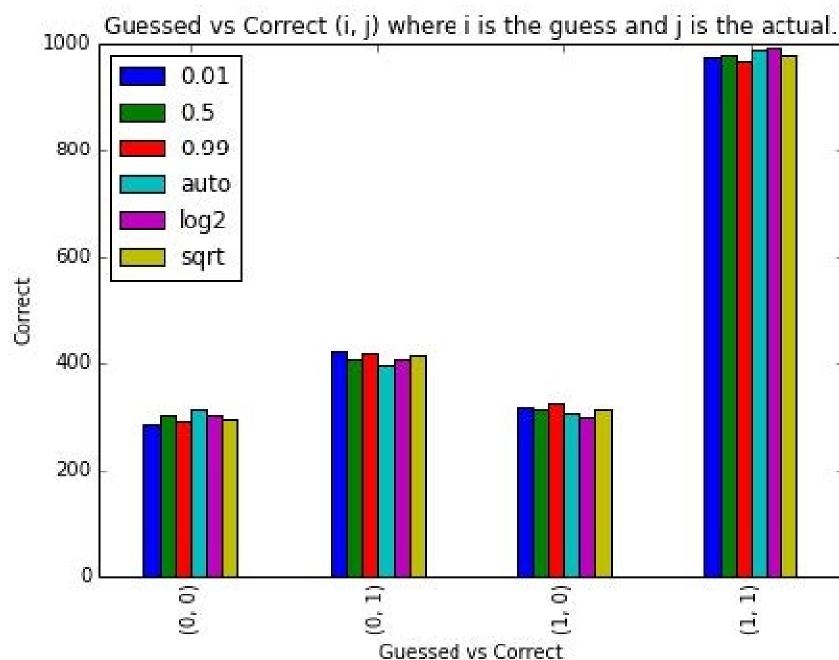
>>> confusion_df.plot(kind='bar', ax=ax)

>>> ax.legend(loc='best')

>>> ax.set_title("Guessed vs Correct (i, j) where i is the guess
 and j is the actual.")

>>> ax.grid()

>>> ax.set_xticklabels([str((i, j)) for i, j list(itertools.prod
uct(range(2), range(2)))]);
>>> ax.set_xlabel("Guessed vs Correct")
>>> ax.set_ylabel("Correct")
```

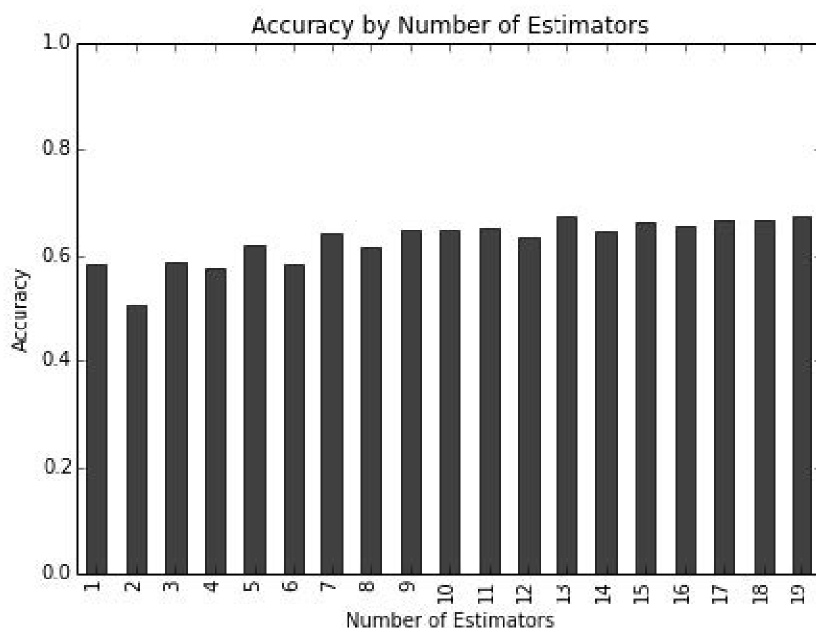


虽然我们看不到表现中的任何真正的差异，对于你自己的项目来说，这是个非常简单的过程。让我们试一试 `n_estimator` 选项，但是使用原始的精确度。使用一些更多的选项，我们的图表就会变得很密集，难以使用。

由于我们正在使用混淆矩阵，我们可以从混淆矩阵的迹除以总和来计算准确度。

```
>>> n_estimator_params = range(1, 20)
>>> confusion_matrixes = {}
>>> for n_estimator in n_estimator_params:
        rf = RandomForestClassifier(n_estimators=n_estimator)
        rf.fit(X[training], y[training])
        confusion_matrixes[n_estimator] = confusion_matrix(y[~tr
aining],
                                          rf.predict(X[~training
]))
        # here's where we'll update the confusion matrix with th
e operation we talked about
>>> accuracy = lambda x: np.trace(x) / np.sum(x, dtype=float)
>>> confusion_matrixes[n_estimator] = accuracy(confusion_matrixe
s[n_estimator])
>>> accuracy_series = pd.Series(confusion_matrixes)
>>> import itertools
>>> from matplotlib import pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> accuracy_series.plot(kind='bar', ax=ax, color='k', alpha=.75
)
>>> ax.grid()
>>> ax.set_title("Accuracy by Number of Estimators")
>>> ax.set_ylim(0, 1) # we want the full scope
>>> ax.set_ylabel("Accuracy")
>>> ax.set_xlabel("Number of Estimators")
```

输出如下：



现在对于大多数部分，准确度是如何上升的呢？确实有一些随机性和准确度相关，但是图像从左到右是上升的。在下面的工作原理一节，我们会讨论随机森林和 bootstrap 之间的关系，以及哪个通常更好。

工作原理

bootstarp 是个不错的技巧，用于扩展模型的其它部分。通常用于介绍 bootstarp 的案例是将标准误差与中位数相加。这里，我们估算了结果，并将估算聚集为概率。

所以，通过简单增加估算器的数量，我们增加了子样本，产生了整体上更快的收敛。

更多

我们可能打算加快训练过程。我之前提到了这个过程，但是同时，我们可以将 `n_jobs` 设为我们想要训练的树的数量。这应该大致等于机器的核数。

```
>>> rf = RandomForestClassifier(n_jobs=4, verbose=True)
>>> rf.fit(X, y)
[Parallel(n_jobs=4)]: Done   1 out of   4 | elapsed:   0.3s remaini
ng:  0.9s
[Parallel(n_jobs=4)]: Done   4 out of   4 | elapsed:   0.3s finishe
d
```

这也可以并行预测：

```
>>> rf.predict(X)
[Parallel(n_jobs=4)]: Done   1 out of   4 | elapsed:   0.0s remaini
ng:    0.0s
[Parallel(n_jobs=4)]: Done   4 out of   4 | elapsed:   0.0s finishe
d

array([1, 1, 0, ..., 1, 1, 1])
```

4.5 使用支持向量机对数据分类

支持向量机（SVM）是我们使用的技巧之一，它不能轻易用概率解释。SVM 背后的原理是，我们寻找一个平面，，它将数据集分割为组，并且是最优的。这里，分割的意思是，平面的选择使平面上最接近的点之间的间距最大。这些点叫做支持向量。

准备

SVM 是我最喜欢的机器学习算法之一，它是我在学校中学习的第一批机器学习伏安法之一。所以，让我们获得一些数据并开始吧。

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification()
```

操作步骤

创建支持向量机分类器的机制非常简单。有许多可用的选项。所以，我们执行下列操作：

1. 创建 SVC 对象，并训练一些伪造数据。
2. 使用 SVC 对象训练一些示例数据。
3. 稍微讨论一些 SVC 选项。

从支持向量机模块导入支持向量分类器（SVC）：

```
>>> from sklearn.svm import SVC
>>> base_svm = SVC()
>>> base_svm.fit(X, y)
```

让我们看一些属性：

- `C` ：以防我们的数据集不是分离好的， `C` 会在间距上放大误差。随着 `C` 变大，误差的惩罚也会变大，SVM 会尝试寻找一个更窄的间隔，即使它错误分类了更多数据点。
- `class_weight` ：这个表示问题中的每个类应该给予多少权重。这个选项以字典提供，其中类是键，值是与这些类关联的权重。
- `gamma` ：这是用于核的 Gamma 参数，并且由 `rgb, sigmoid` 和 `ploy` 支持。
- `kernel` ：这是所用的核，我们在下面使用 `linear` 核，但是 `rgb` 更流行，并且是默认选项。

工作原理

我们在准备一节中说过，SVM 会尝试寻找一个屏幕，它使两个类别最优分割。让我们查看带有两个特征的最简单示例，以及一个良好分割的结果。

首先，让我们训练数据集，之后我们将其绘制出来。

```
>>> X, y = datasets.make_blobs(n_features=2, centers=2)
>>> from sklearn.svm import LinearSVC
>>> svm = LinearSVC()
>>> svm.fit(X, y)
```

既然我们训练了支持向量机，我们将图中每个点结果绘制出来。这会向我们展示近似的决策边界。

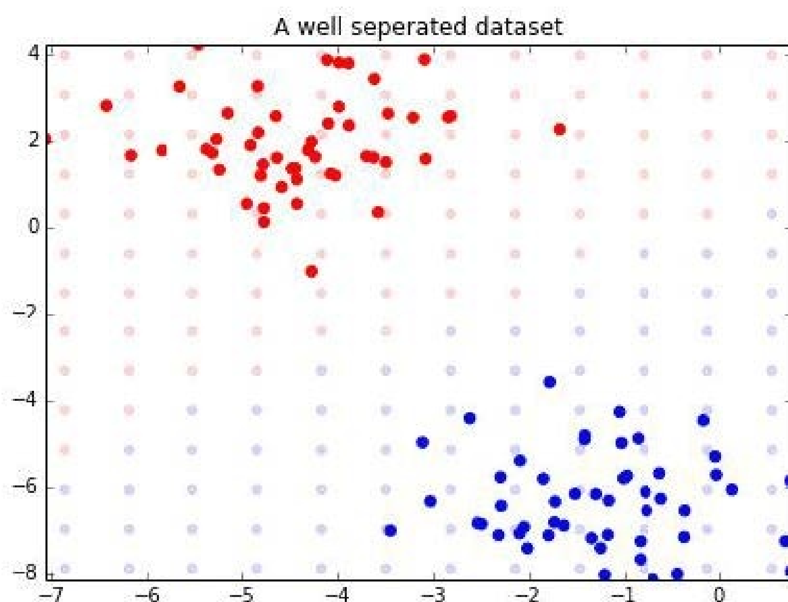
```
>>> from itertools import product
>>> from collections import namedtuple

>>> Point = namedtuple('Point', ['x', 'y', 'outcome'])
>>> decision_boundary = []
>>> xmin, xmax = np.percentile(X[:, 0], [0, 100])
>>> ymin, ymax = np.percentile(X[:, 1], [0, 100])

>>> for xpt, ypt in product(np.linspace(xmin-2.5, xmax+2.5, 20),
    np.linspace(ymin-2.5, ymax+2.5, 20)):
        p = Point(xpt, ypt, svm.predict([xpt, ypt]))
            decision_boundary.append(p)

>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> for xpt, ypt, pt in decision_boundary:
        ax.scatter(xpt, ypt, color=colors[pt[0]], alpha=.15)
        ax.scatter(X[:, 0], X[:, 1], color=colors[y], s=30)
        ax.set_ylim(ymin, ymax)
        ax.set_xlim(xmin, xmax)
        ax.set_title("A well separated dataset")
```

输出如下：



让我们看看其他例子，但是这一次决策边界不是那么清晰：

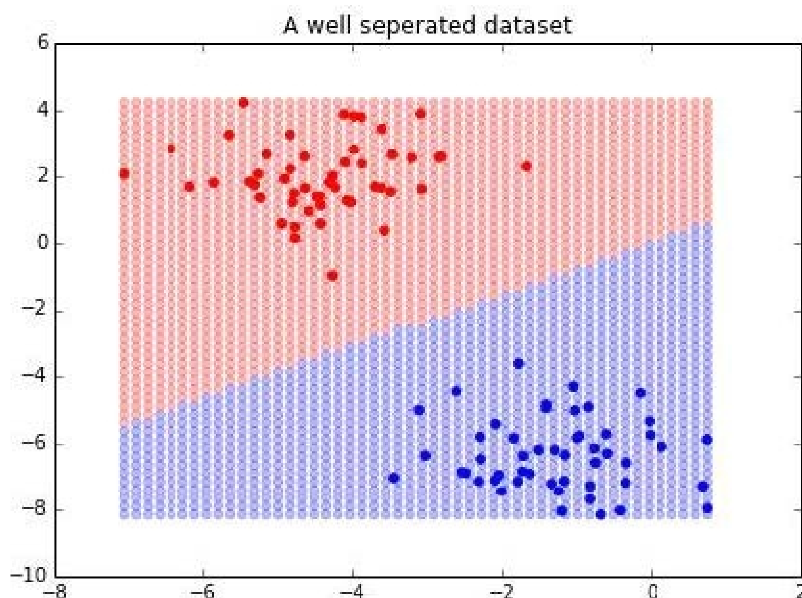
```
>>> X, y = datasets.make_classification(n_features=2,
                                        n_classes=2,
                                        n_informative=2,
                                        n_redundant=0)
```

我们已经看到，这并不是用线性分类易于解决的问题。

虽然我们不会将其用于实战，让我们看一看决策边界。首先，让我们使用新的数据点重新训练分类器。

```
>>> svm.fit(X, y)
>>> xmin, xmax = np.percentile(X[:, 0], [0, 100])
>>> ymin, ymax = np.percentile(X[:, 1], [0, 100])
>>> test_points = np.array([[xx, yy] for xx, yy in
                       product(np.linspace(xmin, xmax),
                       np.linspace(ymin, ymax))])
>>> test_preds = svm.predict(test_points)
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> ax.scatter(test_points[:, 0], test_points[:, 1],
           color=colors[test_preds], alpha=.25)
>>> ax.scatter(X[:, 0], X[:, 1], color=colors[y])
>>> ax.set_title("A well separated dataset")
```

输出如下：



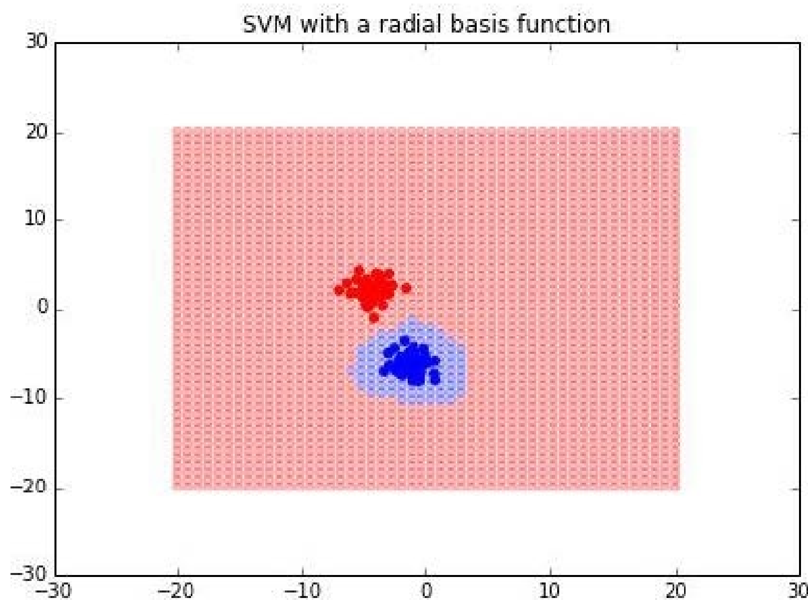
我们可以看到，决策边界并不完美，但是最后，这是我们获得的最好的线性 SVM。

更多

随让我们可能不能获得更好的线性 SVM，Scikit 中的 SVC 分类器会使用径向基函数。我们之前看过这个函数，但是让我们观察它如何计算我们刚刚拟合的数据集的决策边界。

```
>>> radial_svm = SVC(kernel='rbf')
>>> radial_svm.fit(X, y)
>>> xmin, xmax = np.percentile(X[:, 0], [0, 100])
>>> ymin, ymax = np.percentile(X[:, 1], [0, 100])
>>> test_points = np.array([[xx, yy] for xx, yy in
                            product(np.linspace(xmin, xmax),
                            np.linspace(ymin, ymax))])
>>> test_preds = radial_svm.predict(test_points)
>>> import matplotlib.pyplot as plt
>>> f, ax = plt.subplots(figsize=(7, 5))
>>> import numpy as np
>>> colors = np.array(['r', 'b'])
>>> ax.scatter(test_points[:, 0], test_points[:, 1],
               color=colors[test_preds], alpha=.25)
>>> ax.scatter(X[:, 0], X[:, 1], color=colors[y])
>>> ax.set_title("SVM with a radial basis function")
```

输出如下：



我们可以看到，决策边界改变了。我们甚至可以传入我们自己的径向基函数，如果需要的话：

```
>>> def test_kernel(X, y):
        """ Test kernel that returns the exponentiation of the d
ot of the
        X and y matrices.
        This looks an awful lot like the log hazards if you're f
amiliar with survival analysis.
        """
        return np.exp(np.dot(X, y.T))
>>> test_svc = SVC(kernel=test_kernel)
>>> test_svc.fit(X, y)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, degree=3
,
    gamma=0.0,   kernel=<function test_kernel at 0x121fdfb90>,
    max_iter=-1,   probability=False, random_state=None,
    shrinking=True, tol=0.001,   verbose=False)
```

4.6 使用多类分类来归纳

这个秘籍中，我们会了解多类分类。取决于你的算法选择，你可以轻松地实现多类分类，或者定义用于比较的模式。

准备

在处理线性模型，例如逻辑回归时，我们需要使用 `OneVsRestClassifier` 。这个模式会为每个类创建一个分类器。

操作步骤

首先，我们会查看一个决策树的粗略示例，用于拟合多类的数据集。我们之前讨论过，我们可以使用一些分类器获得多类，所以我们仅仅拟合示例来证明它可以工作，然后继续。

其次，我们实际上将 `OneVsRestClassifier` 合并进我的模型中：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=10000, n_class
es=3,
                                        n_informative=3)
>>> from sklearn.tree import DecisionTreeClassifier
>>> dt = DecisionTreeClassifier()
>>> dt.fit(X, y)
>>> dt.predict(X)
array([1, 1, 0, .., 2, 1, 1])
```

你可以看到，我们能够以最低努力来拟合分类器。

现在，让我们转向多类分类器的案例中。这需要我们导入 `OneVsRestClassifier` 。我们也导入 `LogisticRegression` 。

```
>>> from sklearn.multiclass import OneVsRestClassifier
>>> from sklearn.linear_model import LogisticRegression
```

现在，我们覆盖 `LogisticRegression` 分类器，同时，注意我们可以使其并行化。如果我们想知道 `OneVsRestClassifier` 分类器如何工作，它仅仅是训练单独的模型，之后比较它们。所以，我们可以同时单独训练数据。

```
>>> mlr = OneVsRestClassifier(LogisticRegression(), n_jobs=2)
>>> mlr.fit(X, y)
>>> mlr.predict(X)
array([1, 1, 0, ..., 2, 1, 1])
```

工作原理

如果我们打算快速时间我们自己的 `OneVsRestClassifier` ，应该怎么做呢？

首先，我们需要构造一种方式，来迭代分类，并为每个分类训练分类器。之后，我们首先需要预测每个分类：

```
>>> import numpy as np
>>> def train_one_vs_rest(y, class_label):
        y_train = (y == class_label).astype(int)
        return y_train
>>> classifiers = []
>>> for class_i in sorted(np.unique(y)):       l = LogisticRegre
ssion()
        y_train = train_one_vs_rest(y, class_i)
        l.fit(X, y_train)
        classifiers.append(l)
```

好的，所以既然我们配置好了 OneVsRest 模式，我们需要做的所有事情，就是求出每个数据点对于每个分类器的可能性。我们之后将可能性最大的分类赋给数据点。

例如，让我们预测 `X[0]` ：

```
for classifier in classifiers
>>>  print classifier.predict_proba(X[0])

[[ 0.90443776  0.09556224]]
[[ 0.03701073  0.96298927]]
[[ 0.98492829  0.01507171]]
```

你可以看到，第二个分类器（下标为 `1` ）拥有“正”的最大可能性，所以我们将这个点标为 `1` 。

4.7 将 LDA 用于分类

线性判别分析（LDA）尝试拟合特征的线性组合，来预测结果变量。LDA 通常用作预处理步骤，我们会在这篇秘籍中涉及这两种方法。

准备

这篇秘籍中，我们会做这些事情：

1. 从雅虎获取股票数据
2. 将其重新排列为我们熟悉的形状
3. 创建 LDA 对象来拟合和预测类标签
4. 给出如何使用 LDA 来降维的示例

操作步骤

这个例子中，我们就执行一种分析，类似于 Altman 的 Z 规范化。在他的论文中，Altamn 基于多种金融度量，预测了公司两年内的违约可能性。从它的维基页面中，我们看到了下面这些东西：

T1 = 流动资产 / 总资产 。相对于公司规模来度量流动资产 。

T2 = 留存收益 / 总资产 。度量反映公司年龄和盈利能力的收益率。

T3 = 利息和税前的收益 / 总资产 。度量税和杠杆系数之外的运作效率。它是运作收益，对于长期生存非常重要。

T4 = 股权市值 / 总负债的账面价值。添加市场维度，可以将证券价格波动展示为危险信号。

T5 = 营业额 / 总资产。总资产周转的标准度量（每个产业的变化都很大）。

这段来自维基百科：

[1] Altman, Edward I. (September 1968). ""Financial Ratios, Discriminant Analysis and the Prediction of Corporate Bankruptcy"". Journal of Finance: 189–209.

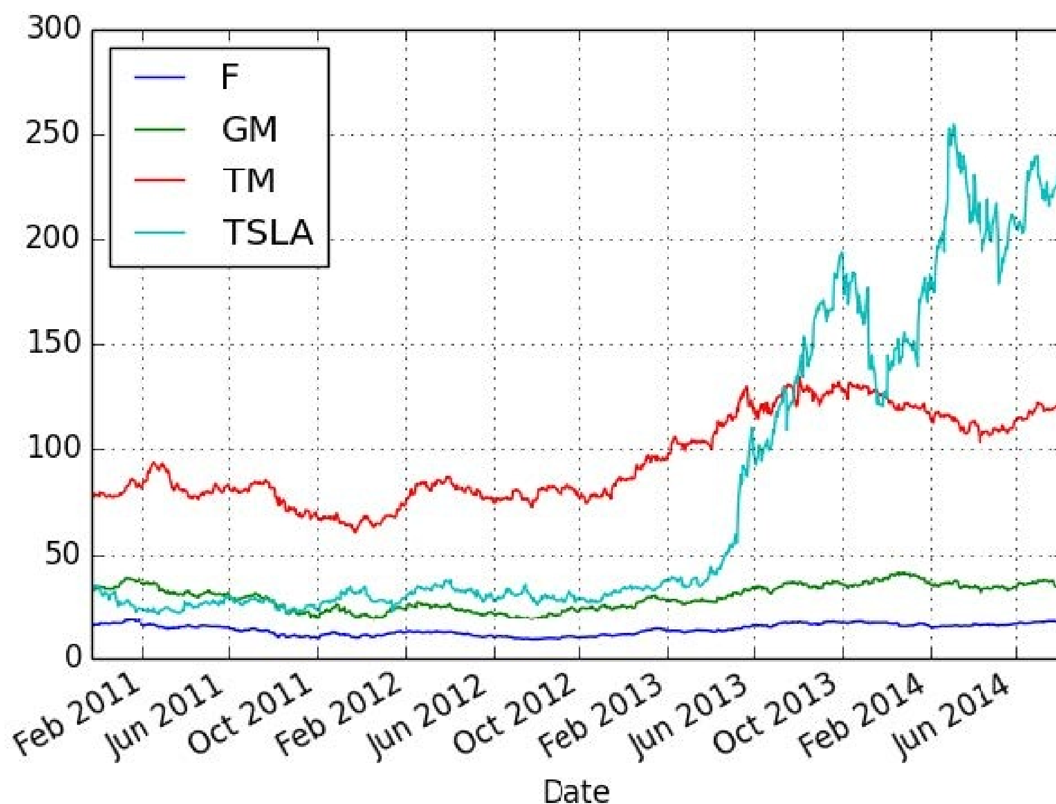
这个分析中，我们使用 Pandas 从雅虎抓取一些金融数据。我们尝试预测，股票是否在六个月内会涨，基于股票的当前属性。很显然没有比 Altman 的 Z 规范化更精妙的东西了。让我们使用一些汽车股票：

```
>>> tickers = ["F", "TM", "GM", "TSLA"]
>>> from pandas.io import data as external_data
>>> stock_panel = external_data.DataReader(tickers, "yahoo")
```

这个数据结构是 `panel`，类似于 OLAP 立方体，或者三维的 `DataFrame`。让我们看看这些数据，来熟悉一下收盘价，因为我们在比较时只考虑它。

```
>>> stock_df = stock_panel.Close.dropna()
>>> stock_df.plot(figsize=(7, 5))
```

下面是输出：



好的，所以现在我们需要将每支股票的价格和它六个月内的价格比较。如果更高，置为 1，否则置为 0。

为此，我们只需要将其向前移动 180 天并比较：

```
#this dataframe indicates if the stock was higher in 180 days
>>> classes = (stock_df.shift(-180) > stock_df).astype(int)
```

我们需要做的下一件事，就是展开数据集：

```
>>> X = stock_panel.to_frame()
>>> classes = classes.unstack()
>>> classes = classes.swaplevel(0, 1).sort_index()
>>> classes = classes.to_frame()
>>> classes.index.names = ['Date', 'minor']
>>> data = X.join(classes).dropna()
>>> data.rename(columns={0: 'is_higher'}, inplace=True)
>>> data.head()
```

输出如下：

		Open	High	Low	Close	Volume	Adj Close	is_higher
Date	minor							
2010-11-18	F	16.77	16.87	16.05	16.12	256937900	15.07	0
	GM	35.00	35.99	33.89	34.19	457044300	33.61	0
	TM	77.36	77.51	76.83	77.29	989100	77.29	0
	TSLA	30.67	30.74	28.92	29.89	956100	29.89	0
2010-11-19	F	16.02	16.38	15.83	16.28	130323600	15.22	0

好的，我们现在需要为 SciPy 创建矩阵。为此，我们会使用 `patsy` 库。这是一个非常棒的库，可以用于创建和 R 相似的决策矩阵。

```
>>> import patsy
>>> X = patsy.dmatrix("Open + High + Low + Close + Volume +
                       is_higher - 1", data.reset_index(),
                       return_type='dataframe')
>>> X.head()
```

输出如下：

	Open	High	Low	Close	Volume	is_higher
0	16.77	16.87	16.05	16.12	256937900	0
1	35.00	35.99	33.89	34.19	457044300	0
2	77.36	77.51	76.83	77.29	989100	0
3	30.67	30.74	28.92	29.89	956100	0
4	16.02	16.38	15.83	16.28	130323600	0

`patsy` 是个非常强大的包。例如，假设我们打算使用第一章的一些预处理。在 `patsy` 中，可以像 R 一样，修改公式相当于修改决策矩阵。这里并不会这么做，但是如果我们打算将数据缩放为均值 0 和标准差 1，函数就是 `scale(open) + scale(high)` 。

太棒了。所以现在我们得到了数据集。让我们训练 LDA 对象吧。

```
>>> import pandas as pd
>>> from sklearn.lda import LDA
>>> lda = LDA()
>>> lda.fit(X.ix[:, :-1], X.ix[:, -1]);
```

我们可以看到，数据集的预测并不是非常糟糕。确实，我们打算使用其他参数改进并测试模型：

```
>>> from sklearn.metrics import classification_report
>>> print classification_report(X.ix[:, -1].values,
                                lda.predict(X.ix[:, :-1]))
             precision    recall  f1-score   support
0.0               0.63      0.59      0.61      1895
1.0               0.60      0.64      0.62      1833
avg / total       0.61      0.61      0.61      3728
```

这些度量以多种方式描述了模型如何拟合数据。

- 对于 `precision`（准确率），假设模型预测正值，多大比例这个是对的？（ `TP / (TP + FP)` ）
- 对于 `recall`（召回率），假设某个类的状态是正确的，我们选取了多大的比例？由于召回率是搜索问题中的常见度量，所以我用“选取”。例如，有一些潜在的网页和搜索术语相关，也就是返回的比例。（ `TP / (TP + FN)` ）
- `f1-score` 参数尝试总结二者关系。（ `2 * p * r / (p + r)` ）

工作原理

LDA 实际上非常类似于我们之前做过的聚类。我们从数据训练线性模型。之后，一旦我们拥有了模型，我们尝试预测并比较每个类的数据的可能性。我们选择更加常见的选项。

LDA 实际上是 QDA 的简化，我们会在下一节谈到它。这里，我们假设每个类的协方差都是一样的，但是 QDA 中，这个假设是不严格的。可以将它们的关系类比为 KNN 和 GMM。

4.8 使用 QDA - 非线性 LDA

QDA 是一些通用技巧的推广，例如平方回归。它只是模型的推广，能够拟合更复杂的模型。但是，就像其它东西那样，当混入复杂性时，就更加困难了。

准备

我们会扩展上一个秘籍，并通过 QDA 对象查看平方判别分析（QDA）。

我们说过我们会根据模型的协方差做出假设。这里，我们会缓和这个假设。

操作步骤

QDA 是 `qda` 模块的一个成员。使用下列命令来使用 QDA：

```
>>> from sklearn.qda import QDA
>>> qda = QDA()

>>> qda.fit(X.ix[:, :-1], X.ix[:, -1])
>>> predictions = qda.predict(X.ix[:, :-1])
>>> predictions.sum()
2812.0

>>> from sklearn.metrics import classification_report
>>> print classification_report(X.ix[:, -1].values, predictions)

              precision    recall  f1-score   support
0.0                0.75      0.36      0.49      1895
1.0                0.57      0.88      0.69      1833
avg / total        0.66      0.62      0.59      3728
```

你可以看到，总体来说都差不多。如果我们回顾 LDA 秘籍，我们可以看到，与 QDA 相比，类 0 有很大变化，类 1 变化很小。

工作原理

我们在上一个秘籍中提到过，我们本质上在这里比较可能性。所以，如何比较可能性呢？让我们使用价格来尝试对 `is_higher` 分类。

我们假设收盘价服从对数正态分布。为了计算每个类的可能性，我们需要为每个类创建收盘价的子集，以及训练集和测试集。作为下一章的前瞻，我们使用内建的交叉验证方法：

```
>>> from sklearn import cross_validation as cv
>>> import scipy.stats as sp
>>> for test, train in cv.ShuffleSplit(len(X.Close), n_iter=1):
        train_set = X.iloc[train]
        train_close = train_set.Close

        train_0 = train_close[~train_set.is_higher.astype(bool)]

        train_1 = train_close[train_set.is_higher.astype(bool)]


        test_set = X.iloc[test]
        test_close = test_set.Close.values

        ll_0 = sp.norm.pdf(test_close, train_0.mean())
        ll_1 = sp.norm.pdf(test_close, train_1.mean())
```

我们有了两个类的可能性，我们可以比较和分配类：

```
>>> (ll_0 > ll_1).mean()
0.15588673621460505
```

4.9 使用随机梯度下降来分类

我们在第二章中讨论过，随机梯度下降是个用于训练分类模型的基本技巧。这两种技巧之间有一些自然联系，因为名称就暗示了这一点。

准备

在回归中，我们最小化了损失函数，它用于惩罚连续刻度上的不良选择。但是对于分类，我们会最小化损失函数，它用于乘法两个或更多情况。

操作步骤

首先，让我们创建一些基本数据：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification()
```

然后创建 `SGDClassifier` 实例：

```
>>> from sklearn import linear_model
>>> sgd_clf = linear_model.SGDClassifier()
```

像往常一样，我们训练模型：

```
>>> sgd_clf.fit(X, y)
SGDClassifier(alpha=0.0001, class_weight=None, epsilon=0.1, eta0=
0.0,
              fit_intercept=True, l1_ratio=0.15,
              learning_rate='optimal', loss='hinge', n_iter=5,
              n_jobs=1, penalty='l2', power_t=0.5,
              random_state=None,
              shuffle=False, verbose=0, warm_start=False)
```

我们可以设置 `class_weight` 参数来统计数据集中不平衡的变化总数。

Hinge 损失函数定义为：

```
max(0, 1 - ty)
```

这里， `t` 是真正分类，+1 为一种情况，-1 为另一种情况。系数向量记为 `y` ，因为它是从模型中拟合出来的。 `x` 是感兴趣的值。这也是一种很好的度量方式。以另外一种形式表述：

```
t ∈ -1, 1
y = βx + b
```

4.10 使用朴素贝叶斯来分类数据

朴素分页四是个非常有意思的模型。它类似于 KNN，做了一些假设来简化事实，但是仍然在许多情况下都很好。

准备

这个秘籍中，我们会使用朴素贝叶斯来分类文档。我拥有个人经验的一个示例就是，使用会计学中的组成账户描述符的单词，例如应付账款，来判断它属于利润表、现金流转表、还是资产负债表。

基本理念是使用来自带标签的测试语料库中的词频，来学习文档的分类。之后，我们可以将其用在训练集上来尝试预测标签。

我们使用 Sklearn 中的 `newgroups` 数据集来玩转朴素贝叶斯模型。这是有价值的一组数据，所以我们抓取它而不是加载它。我们也将分类限制为 `rec.autos` 和 `rec.motorcycles` 。

```
>>> from sklearn.datasets import fetch_20newsgroups

>>> categories = ["rec.autos", "rec.motorcycles"]
>>> newgroups = fetch_20newsgroups(categories=categories)

#take a look
>>> print "\n".join(newgroups.data[:1])
From: gregl@zimmer.CSUFresno.EDU (Greg Lewis)
Subject: Re: WARNING.....(please read)...
Keywords: BRICK, TRUCK, DANGER
Nntp-Posting-Host: zimmer.csufresno.edu
Organization: CSU Fresno
Lines: 33

[…]

>>> newgroups.target_names[newgroups.target[:1]]
'rec.autos'
```

既然我们拥有了 `newgroups` ，我们需要将每个文档表示为词频向量。这个表示就是朴素贝叶斯名称的来历。模型是“朴素”的，不按照任何文档间的单词协方差，来对文档进行分类。这可以认为是可以缺陷，但是朴素贝叶斯已经被证实相当可靠。

我们需要将数据处理为词频矩阵。这是个稀疏矩阵，当某个单词出现在文档中时，这个单词就有条目。这个矩阵可能非常大，就像这样：

```
>>> from sklearn.feature_extraction.text import CountVectorizer

>>> count_vec = CountVectorizer()
>>> bow = count_vec.fit_transform(newgroups.data
```

这个矩阵是个稀疏矩阵，它的长度是文档数量乘以不同单词的数量。它的值是每个文档中每个单词的频率。

```
>>> bow <1192x19177 sparse matrix of type '<type 'numpy.int64'>'
    with 164296 stored elements in Compressed Sparse Row format>
```

我们实际上需要将矩阵表示为密集数组，用于朴素贝叶斯对象。所以，让我们将其转换回来。

```
>>> bow = np.array(bow.todense())
```

显然，多数元素都是 0，但是我们可能打算重构文档的统计，作为合理性检查。

```
>>> words = np.array(count_vec.get_feature_names())
>>> words[bow[0] > 0][:5]
array([u'10pm', u'1qh336innfl5', u'33', u'93740',
u'_______________________________________________________________
_____'],
dtype='<U79')
```

现在，这些就是第一个文档的示例了？让我们使用下面的命令：

```
>>> '10pm' in newgroups.data[0].lower()
True
>>> '1qh336innfl5' in newgroups.data[0].lower()
True
```

操作步骤

好的，所以需要比平常更多的时间来准备数据，因为我们处理的文本数据，并不是能够像之前的矩阵那样快速表示。

但是，既然我们准备好了，我们启动分类器来训练我们的模型。

```
>>> from sklearn import naive_bayes
>>> clf = naive_bayes.GaussianNB()
```

在我们训练模型之前，让我们将数据集划分为训练集和测试集。

```
>>> mask = np.random.choice([True, False], len(bow))
>>> clf.fit(bow[mask], newgroups.target[mask])
>>> predictions = clf.predict(bow[~mask])
```

既然我们在训练集训练了模型，之后预测测试集来尝试判断文章属于哪个分类，让我们获取准确率。

```
>>> np.mean(predictions == newgroups.target[~mask])
0.92446043165467628
```

工作原理

朴素贝叶斯的基本原理，就是我们可以根据特征向量，来估计数据点属于分类的概率（ `P(Ci|X)` ）。

这可以使用贝叶斯定期来变形，来变成特征向量的后验概率（ `P(X|Ci)` ）。如果特征向量的概率最大，那么后延估计就选择这个分类。

更多

我们也可以将朴素贝叶斯扩展来执行多类分类。我们不适用高斯可能性，而是使用多项式可能性。

首先，让我们获取第三个分类：

```
>>> from sklearn.datasets import fetch_20newsgroups
>>> mn_categories = ["rec.autos", "rec.motorcycles",
                     "talk.politics.guns"]
>>> mn_newgroups = fetch_20newsgroups(categories=mn_categories)
```

我们需要将这些东西向量化。

```
>>> mn_bow = count_vec.fit_transform(mn_newgroups.data)
>>> mn_bow = np.array(mn_bow.todense())
```

让我们创建为训练集和测试集创建一个屏蔽数组。

```
>>> mn_mask = np.random.choice([True, False], len(mn_newgroups.d
ata))
>>> multinom = naive_bayes.MultinomialNB()
>>> multinom.fit(mn_bow[mn_mask], mn_newgroups.target[mn_mask])

>>> mn_predict = multinom.predict(mn_bow[~mn_mask])
>>> np.mean(mn_predict == mn_newgroups.target[~mn_mask]) 0.96594
778660612934
```

我们做的很好，完全不会惊讶。我们在两个分类的情况下表现不错，由于 `talk.politics.guns` 分类和其它两个正交，我们应该也表现不错。

4.11 标签传递，半监督学习

标签传递是个半监督学习技巧，它利用带标签和不带标签的数据，来了解不带标签的数据。通常，受益于分类算法的数据是难以标注的。例如，标注数据的开销可能非常大，所以手动标注一个子集边角高效。也就是说，对于公司雇佣分类学家来说，存在可能较慢，但是在发展的支持。

准备

另一个问题范围就是截尾数据。你可以想象一种情况，其中时间的边界会影响你收集数据的能力。也就是说，例如，你将试验药物给病人，并测量它们。有些时候，你能够测量药物的结果。如果碰巧足够快，但是你可能打算预测药物的结果，它们

的反应时间较慢。这些药物可能对一些病人有致命的反应，并且需要采取救生措施。

操作步骤

为了表示半监督或者截尾数据，我们需要做一些简单的数据处理。首先，我们会浏览一个简单的示例，之后转向一些更加困难的情况。

```
>>> from sklearn import datasets
>>> d = datasets.load_iris()
```

由于我们会将数据搞乱，我们做一个备份，并向标签名称数组的副本添加一个 `unlabeled` 成员。它会使数据的识别变得容易。

```
>>> X = d.data.copy()
>>> y = d.target.copy()
>>> names = d.target_names.copy()
>>> names = np.append(names, ['unlabeled'])
>>> names
array(['setosa', 'versicolor', 'virginica', 'unlabeled'],
       dtype='|S10')
```

现在使用 `-1` 更新 `y`，这就是未标注情况的记号。这也是我们将 `unlabeled` 添加到末尾的原因。

```
>>> y[np.random.choice([True, False], len(y))] = -1
```

我们的数据现在拥有一系列负值（`-1`），散布在真正数据当中：

```
>>> y[:10]
array([-1, -1, -1, -1,  0,  0, -1, -1,  0, -1])
```

```
>>> names[y[:10]]
array(['unlabeled', 'unlabeled', 'unlabeled', 'unlabeled', 'seto
sa',
       'setosa', 'unlabeled', 'unlabeled', 'setosa', 'unlabeled'
],
       dtype='|S10')
```

我们显然拥有一大堆未标注的数据，现在的目标是使用 `LabelPropagation` 来预测标签：

```
>>> from sklearn import semi_supervised
>>> lp = semi_supervised.LabelPropagation()

>>> lp.fit(X, y)

LabelPropagation(alpha=1, gamma=20, kernel='rbf', max_iter=30,
                 n_neighbors=7, tol=0.001)

>>> preds = lp.predict(X)
>>> (preds == d.target).mean()
0.98666666666666669
```

并不是太坏。我们使用了所有数据，所以这是一种作弊。并且，`iris` 数据集是个良好分隔的数据集。

虽然我们完成了，让我们看看 `LabelSpreading`，它是 `LabelPropagation` 的姐妹类。我们会在“工作原理”一节给出 `LabelSpreading` 和 `LabelPropagation` 的技术差异，但是很容易看出它们及其相似。

```
>>> ls = semi_supervised.LabelSpreading()
```

观察它的工作原理，`LabelSpreading` 更加健壮和嘈杂。

```
>>> ls.fit(X, y)
LabelSpreading(alpha=0.2, gamma=20, kernel='rbf', max_iter=30,
               n_neighbors=7, tol=0.001)

>>> (ls.predict(X) == d.target).mean()
0.96666666666666667
```

不要认为标签传播算法丢失了几个指标，它就表现得更差了。关键是，我们可能提供一些预测训练集的能力，并且适用于更广泛的环境。

工作原理

标签传递的原理是，创建数据点的图，每条边上的权重为：

```
wij(θ) = dij / θ^2
```

这个算法之后的原理是，数据点将它们的标签传递给未标记的数据点。这个传递部分由边的权重决定。

边上的权重可以放在转移概率矩阵中。我们可以迭代来估计实际的标签。

第五章 模型后处理

> 作者：Trent Hauck
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0

5.1 K-fold 交叉验证

这个秘籍中，我们会创建交叉验证，它可能是最重要的模型后处理验证练习。我们会在这个秘籍中讨论 k-fold 交叉验证。有几种交叉验证的种类，每个都有不同的随机化模式。K-fold 可能是一种最熟知的随机化模式。

准备

我们会创建一些数据集，之后在不同的在不同的折叠上面训练分类器。值得注意的是，如果你可以保留一部分数据，那是最好的。例如，我们拥有 `N = 1000` 的数据集，如果我们保留 200 个数据点，之后使用其他 800 个数据点之间的交叉验证，来判断最佳参数。

工作原理

首先，我们会创建一些伪造数据，之后测试参数，最后，我们会看看结果数据集的大小。

```
>>> N = 1000
>>> holdout = 200
>>> from sklearn.datasets import make_regression
>>> X, y = make_regression(1000, shuffle=True)
```

既然我们拥有了数据，让我们保留 200 个点，之后处理折叠模式。

```
>>> X_h, y_h = X[:holdout], y[:holdout]
>>> X_t, y_t = X[holdout:], y[holdout:]
>>> from sklearn.cross_validation import KFold
```

K-fold 给了我们一些选项，来选择我们想要多少个折叠，是否让值为下标或者布尔值，是否打算打乱数据集，最后是随机状态（主要出于再现性）。下标实际上会在之后的版本中溢出。假设它为 `True` 。

让我们创建交叉验证对象：

```
>>> kfold = KFold(len(y_t), n_folds=4)
```

现在，我们可以迭代 k-fold 对象：

```
>>> output_string = "Fold: {}, N_train: {}, N_test: {}"

>>> for i, (train, test) in enumerate(kfold):
        print output_string.format(i, len(y_t[train]), len(y_t[test]))

Fold: 0, N_train: 600, N_test: 200
Fold: 1, N_train: 600, N_test: 200
Fold: 2, N_train: 600, N_test: 200
Fold: 3, N_train: 600, N_test: 200
```

每个迭代都应该返回相同的分割大小。

工作原理

可能很清楚，但是 k-fold 的原理是迭代折叠，并保留 `1/n_folds * N` 个数据，其中 `N` 是我们的 `len(y_t)` 。

从 Python 的角度看，交叉验证对象拥有一个迭代器，可以通过 `in` 运算符来访问。通常，对于编写交叉验证对象的包装器来说比较实用，它会迭代数据的子集。例如我们可能拥有一个数据集，它拥有数据点的重复度量，或者我们可能拥有一个病人的数据集，每个病人都拥有度量。

我们打算将它们组合起来，并对其使用 Pandas。

```
>>> import numpy as np
>>> import pandas as pd

>>> patients = np.repeat(np.arange(0, 100, dtype=np.int8), 8)

>>> measurements = pd.DataFrame({'patient_id': patients,
                   'ys': np.random.normal(0, 1, 800)})
```

既然我们拥有了数据，我们仅仅打算保留特定的顾客，而不是数据点。

```
>>> custids = np.unique(measurements.patient_id)
>>> customer_kfold = KFold(custids.size, n_folds=4)

>>> output_string = "Fold: {}, N_train: {}, N_test: {}"

>>> for i, (train, test) in enumerate(customer_kfold):
        train_cust_ids = custids[train]
        training = measurements[measurements.patient_id.isin(
                   train_cust_ids)]
        testing = measurements[~measurements.patient_id.isin(
                   train_cust_ids)]

        print output_string.format(i, len(training), len(testing
))

Fold: 0, N_train: 600, N_test: 200
Fold: 1, N_train: 600, N_test: 200
Fold: 2, N_train: 600, N_test: 200
Fold: 3, N_train: 600, N_test: 200
```

5.2 自动化交叉验证

我们会查看如何使用 Sklearn 自带的交叉验证，但是我们也可以使用一个辅助函数，来自动化执行交叉验证。这类似于 Sklearn 中其它对象，如何被辅助函数和流水线包装。

准备

首先，我们需要创建样例分类器，它可以是任何东西，决策树、随机森林，以及其他。对我们来说，它是随机森林。我们之后会创建数据集，并使用交叉验证函数。

工作原理

首先导入 `ensemble` 模块来开始：

```
>>> from sklearn import ensemble
>>> rf = ensemble.RandomForestRegressor(max_features='auto')
```

好的，所以现在，让我们创建一些回归数据：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_regression(10000, 10)
```

既然我们拥有了数据，我们可以导入 `cross_validation` 模块，并获取我们将要使用的函数：

```
>>> from sklearn import cross_validation

>>> scores = cross_validation.cross_val_score(rf, X, y)

>>> print scores
[ 0.86823874  0.86763225  0.86986129]
```

工作原理

很大程度上，它会委托给交叉验证对象。一个不错的事情是，函数会并行处理交叉验证。

我们可开启详细模式：

```
>>> scores = cross_validation.cross_val_score(rf, X, y, verbose=3
,
             cv=4)

[CV] no parameters to be set
[CV] no parameters to be set, score=0.872866 -   0.7s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.873679 -   0.6s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.878018 -   0.7s
[CV] no parameters to be set
[CV] no parameters to be set, score=0.871598 -   0.6s

[Parallel(n_jobs=1)]: Done   1 jobs       | elapsed:     0.7s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:     2.6s fin
ished
```

我们可以看到，在每次迭代中，我们都调用函数来获得得分。我们也知道了模型如何运行。

同样值得了解是的，我们可以对我们尝试拟合的模型，获取预测得分。我们也会讨论如何创建你自己的评分函数。

5.3 使用 ShuffleSplit 交叉验证

`ShuffleSplit` 是最简单的交叉验证技巧之一。这个交叉验证技巧只是将数据的样本用于指定的迭代数量。

准备

`ShuffleSplit` 是另一个简单的交叉验证技巧。我们会指定数据集中的总元素，并且它会考虑剩余部分。我们会浏览一个例子，估计单变量数据集的均值。这有点类似于重采样，但是它说明了一个原因，为什么我们在展示交叉验证的时候使用交叉验证。

操作步骤

首先，我们需要创建数据集。我们使用 NumPy 来创建数据集，其中我们知道底层的均值。我们会对半个数据集采样，来估计均值，并看看它和底层的均值有多接近。

```
>>> import numpy as np

>>> true_loc = 1000
>>> true_scale = 10
>>> N = 1000

>>> dataset = np.random.normal(true_loc, true_scale, N)

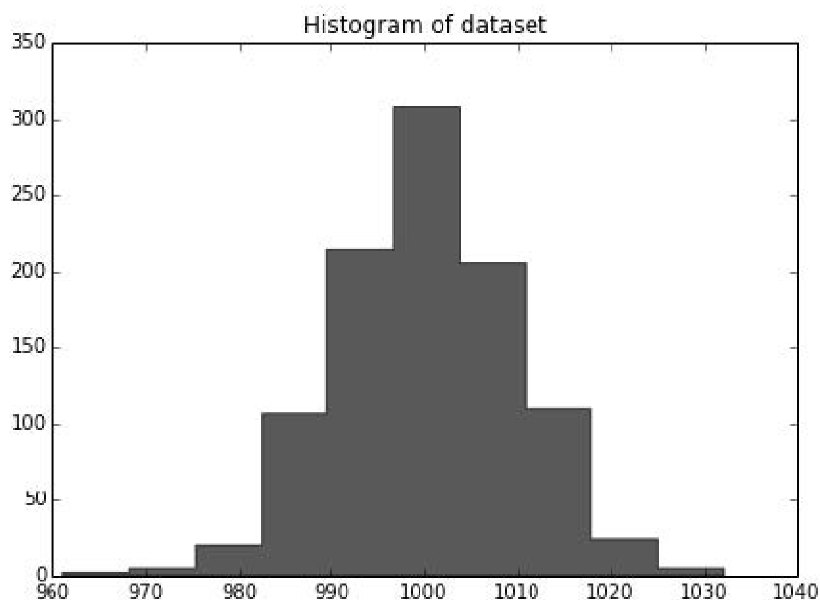
>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.hist(dataset, color='k', alpha=.65, histtype='stepfilled'
);
>>> ax.set_title("Histogram of dataset");

>>> f.savefig("978-1-78398-948-5_06_06.png")
```

NumPy 输出如下：



现在，让我们截取前一半数据集，并猜测均值：

```
>>> from sklearn import cross_validation

>>> holdout_set = dataset[:500]
>>> fitting_set = dataset[500:]

>>> estimate = fitting_set[:N/2].mean()

>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("True Mean vs Regular Estimate")

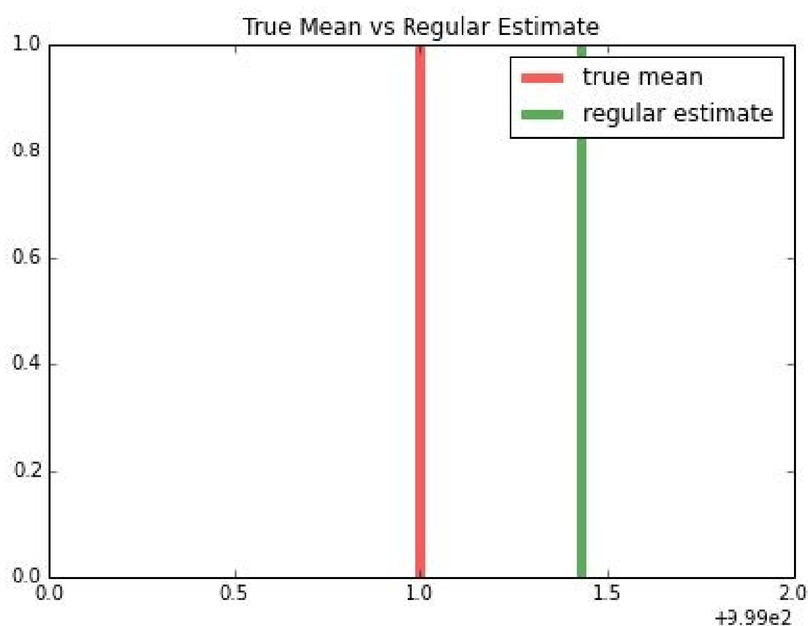
>>> ax.vlines(true_loc, 0, 1, color='r', linestyles='-', lw=5,
              alpha=.65, label='true mean')
>>> ax.vlines(estimate, 0, 1, color='g', linestyles='-', lw=5,
              alpha=.65, label='regular estimate')

>>> ax.set_xlim(999, 1001)

>>> ax.legend()

>>> f.savefig("978-1-78398-948-5_06_07.png")
```

输出如下：



现在，我们可以使用 `ShuffleSplit` 在多个相似的数据集上拟合估计值。

```
>>> from sklearn.cross_validation import ShuffleSplit

>>> shuffle_split = ShuffleSplit(len(fitting_set))

>>> mean_p = []

>>> for train, _ in shuffle_split:
        mean_p.append(fitting_set[train].mean())
        shuf_estimate = np.mean(mean_p)

>>> import matplotlib.pyplot as plt

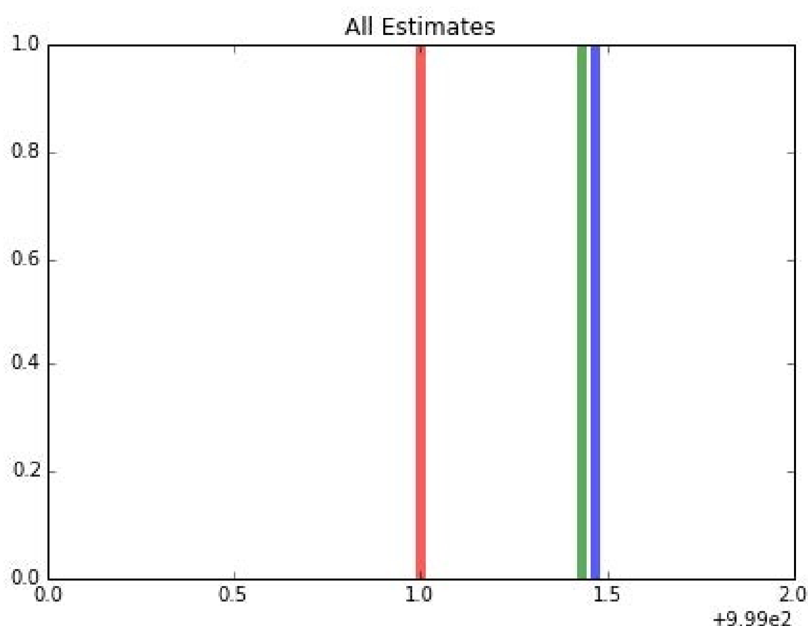
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.vlines(true_loc, 0, 1, color='r', linestyles='-', lw=5,
              alpha=.65, label='true mean')
>>> ax.vlines(estimate, 0, 1, color='g', linestyles='-', lw=5,
              alpha=.65, label='regular estimate')
>>> ax.vlines(shuf_estimate, 0, 1, color='b', linestyles='-', lw=
5,
              alpha=.65, label='shufflesplit estimate')

>>> ax.set_title("All Estimates")
>>> ax.set_xlim(999, 1001)

>>> ax.legend(loc=3)
```

输出如下：



我们可以看到，我们得到了类似于预期的估计值，但是我们可能使用多个样本来获取该值。

5.4 分层的 k-fold

这个秘籍中，我们会快速查看分层的 k-fold 估值。我们会浏览不同的秘籍，其中分类的表示在某种程度上是不平衡的。分层的 k-fold 非常不错，因为他的模式特地为维持分类的比例而设计。

准备

我们打算创建一个小型的数据集。这个数据集中，我们随后会使用分层的 k-fold 验证。我们想让它尽可能小，以便我们查看变化。对于更大的样本，可能并不是特别好。

我们之后会绘制每一步的分类比例，来展示如何维护分类比例。

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=int(1e3),
        weights=[1./11])
```

让我们检查分类的总体权重分布：

```
>>> y.mean()
0.90300000000000002
```

90.5% 的样本都是 1，其余为 0。

操作步骤

让我们创建分层 k-fold 对象，并通过每个折叠来迭代。我们会度量为 1 的 `verse` 比例。之后，我们会通过分割数字来绘制分类比例，来看看是否以及如何发生变化。这个代码展示了为什么它非常好。我们也会对基本的 `ShuffleSplit` 绘制这个代码。

```
>>> from sklearn import cross_validation

>>> n_folds = 50

>>> strat_kfold = cross_validation.StratifiedKFold(y,
                  n_folds=n_folds)
>>> shuff_split = cross_validation.ShuffleSplit(n=len(y),
                  n_iter=n_folds)

>>> kfold_y_props = []
>>> shuff_y_props = []

>>> for (k_train, k_test), (s_train, s_test) in zip(strat_kfold,

>>> shuff_split):
        kfold_y_props.append(y[k_train].mean())
        shuff_y_props.append(y[s_train].mean())
```

现在，让我们绘制每个折叠上的比例：

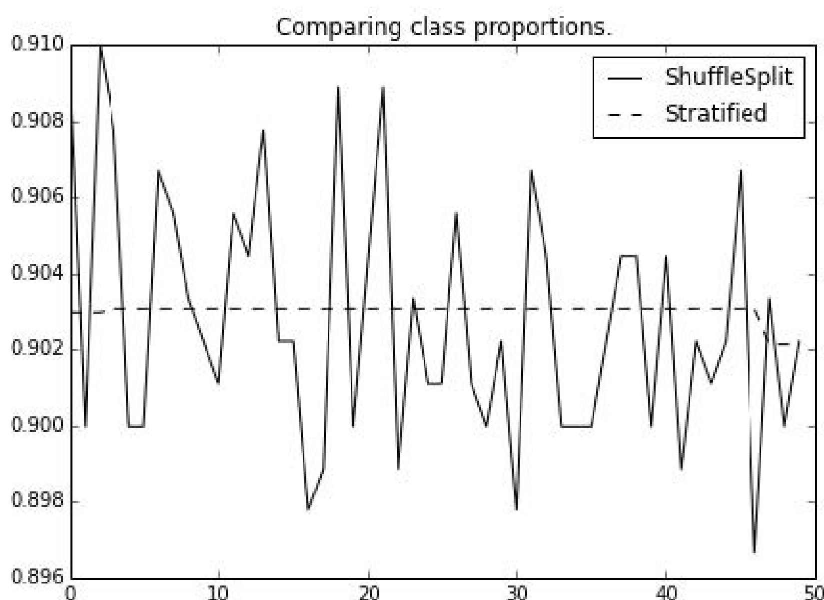
```
>>> import matplotlib.pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.plot(range(n_folds), shuff_y_props, label="ShuffleSplit",
            color='k')
>>> ax.plot(range(n_folds), kfold_y_props, label="Stratified",
            color='k', ls='--')
>>> ax.set_title("Comparing class proportions.")

>>> ax.legend(loc='best')
```

输出如下：



我们可以看到，分层的 k-fold 的每个折叠的比例，在每个折叠之间是稳定的。

工作原理

分层 k-fold 的原理是选取 `y` 值。首先，获取所有分类的比例，之后将训练集和测试集按比例划分。这可以推广到多个标签：

```
>>> import numpy as np

>>> three_classes = np.random.choice([1,2,3], p=[.1, .4, .5],
                          size=1000)

>>> import itertools as it

>>> for train, test in cross_validation.StratifiedKFold(three_cl
asses, 5):
        print np.bincount(three_classes[train])

[  0  90 314 395]
[  0  90 314 395]
[  0  90 314 395]
[  0  91 315 395]
[  0  91 315 396]
```

我们可以看到，我们得到了每个分类的样例大小，正好是训练集合测试集的比例。

5.5 菜鸟的网格搜索

这个秘籍中，我们打算使用 Python 来介绍基本的网格搜索，并且使用 Sklearn 来处理模型，以及 Matplotlib 来可视化。

准备

这个秘籍中，我们会执行下面这些东西：

- 在参数空间中设计基本的搜索网格。
- 迭代网格并检查数据集的参数空间中的每个点的损失或评分函数。
- 选取参数空阿基那种的点，它使评分函数最大或者最小。

同样，我们训练的模型是个基本的决策树分类器。我们的参数空间是 2 维的，有助于我们可视化。

```
criteria = {gini, entropy}
max_features = {auto, log2, None}
```

参数空间是 `criteria` 和 `max_features` 的笛卡尔积。

我们会了解如何使用 `itertools` 来迭代这个空间。

让我们创建数据集来开始：

```
>>> from sklearn import datasets
>>> X, y = datasets.make_classification(n_samples=2000, n_featur
es=10)
```

操作步骤

之前我们说，我们使用网格搜索来调整两个参数 -- `criteria` 和 `max_features``criteria` 和 `max_features` 。我们需要将其表示为 Python 集合，之后使用 `itertools.product` 来迭代它们。

不错，所以既然我们拥有了参数空间，让我们迭代它并检查每个模型的准确率，它们由参数指定。之后，我们保存这个准确率，便于比较不同的参数空间。我们也会使用以 `50, 50` 划分的测试和训练集。

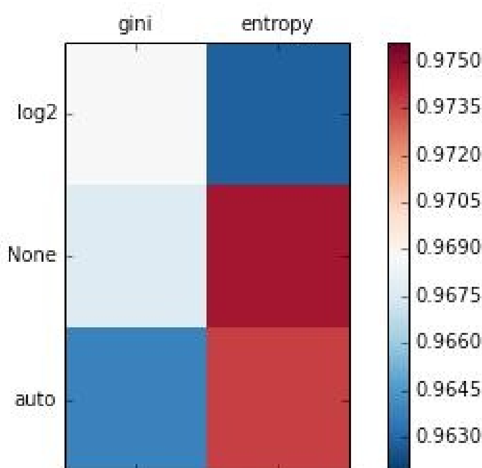
```
import numpy as np
train_set = np.random.choice([True, False], size=len(y))
from sklearn.tree import DecisionTreeClassifier
accuracies = {}
for criterion, max_feature in parameter_space:
    dt = DecisionTreeClassifier(criterion=criterion,

         max_features=max_feature)
    dt.fit(X[train_set], y[train_set])
    accuracies[(criterion, max_feature)] = (dt.predict(X[~train_
set])
                                         == y[~train_set]).mean(
)
>>> accuracies
{('entropy', None): 0.974609375, ('entropy', 'auto'): 0.97363281
25, ('entropy', 'log2'): 0.962890625, ('gini', None): 0.96777343
75, ('gini', 'auto'): 0.9638671875, ('gini', 'log2'): 0.96875}
```

所以现在我们拥有了准确率和它的表现。让我们可视化它的表现。

```
>>> from matplotlib import pyplot as plt
>>> from matplotlib import cm
>>> cmap = cm.RdBu_r
>>> f, ax = plt.subplots(figsize=(7, 4))
>>> ax.set_xticklabels([''] + list(criteria))
>>> ax.set_yticklabels([''] + list(max_features))
>>> plot_array = []
>>> for max_feature in max_features:
        m = []
>>> for criterion in criteria:
        m.append(accuracies[(criterion, max_feature)])
        plot_array.append(m)
>>> colors = ax.matshow(plot_array, vmin=np.min(accuracies.value
s()) -
             0.001, vmax=np.max(accuracies.values()) + 0.001, cm
ap=cmap)
>>> f.colorbar(colors)
```

输出如下：



很容易看到哪个表现最好。单元你可以使用爆破方式看到它如何进一步处理。

工作原理

原理很简单，我们只需要执行下列步骤：

1. 选取一系列参数
2. 迭代它们并求得每一步的准确率
3. 通过可视化来寻找最佳的表现

5.6 爆破网格搜索

这个秘籍中，我们会使用 Sklearn 做一个详细的网格搜索。这基本和上一章的事情相同，但是我们使用内建方法。

我们也会浏览一个执行随机化优化的示例。这是个用于爆破搜索的替代方案。本质上，我们花费一些计算周期，来确保搜索了整个空间。我们在上一个秘籍中比较冷静，但是，你可以想想拥有多个步骤的模型，首先对缺失数据进行估算，之后使用 PCA 降低维度来分类。你的参数空间可能非常大，非常块，因此，搜索一部分空间是有利的。

准备

我们需要下列步骤来开始：

1. 创建一些数据集

2. 之后创建 `LogisticRegression` 对象，训练我们的模型

3. 之后，我们创建搜索对象， `GridSearch` 和 `RandomizedSearchCV`

工作原理

执行下列代码来创建一些分类数据

```
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(1000, n_features=5)
```

现在，我们创建逻辑回归对象：

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(class_weight='auto')
```

我们需要指定打算搜索的参数。对于 `GridSearch` ，我们可以指定所关心的范围，但是对于 `RandomizedSearchCV` ，我们实际上需要指定相同空间上的分布：

```
>>> lr.fit(X, y)

LogisticRegression(C=1.0, class_weight={0: 0.25, 1: 0.75}, dual=
False,
                   fit_intercept=True, intercept_scaling=1,
                   penalty='l2', random_state=None, tol=0.0001)

>>> grid_search_params = {'penalty': ['l1', 'l2'],
                          'C': [1, 2, 3, 4]}
```

我们需要做的唯一一个修改，就是将 `C` 参数描述为概率分布。我们现在使其保持简单，虽然我们使用 `scipy` 来描述这个分布。

```
>>> import scipy.stats as st >>> import numpy as np
>>> random_search_params = {'penalty': ['l1', 'l2'],
                            'C': st.randint(1, 4)}
```

工作原理

现在，我们要训练分类器了。原理是将 `lr` 作为参数传给搜索对象。

```
>>> from sklearn.grid_search import GridSearchCV, RandomizedSear
chCV
>>> gs = GridSearchCV(lr, grid_search_params)
```

`GridSearchCV` 实现了和其他方法相同的 API：

```
>>> gs.fit(X, y)

GridSearchCV(cv=None, estimator=LogisticRegression(C=1.0,
            class_weight='auto', dual=False, fit_intercept=True,
            intercept_scaling=1, penalty='l2', random_state=None
,
            tol=0.0001), fit_params={}, iid=True, loss_func=None
,
            n_jobs=1, param_grid={'penalty': ['l1', 'l2'],
            'C': [1, 2, 3, 4]}, pre_dispatch='2*n_jobs', refit=T
rue,
            score_func=None, scoring=None, verbose=0)
```

我们可以看到，`param_grid` 参数中的 `penalty` 和 `C` 都是数组。

为了评估得分，我们可以使用网格搜索的 `grid_scores_` 属性。我们也打算寻找参数的最优集合。我们也可以查看网格搜索的边际表现。

```
>>> gs.grid_scores_
[mean: 0.90300, std: 0.01192, params: {'penalty': 'l1', 'C': 1},

 mean: 0.90100, std: 0.01258, params: {'penalty': 'l2', 'C': 1},

 mean: 0.90200, std: 0.01117, params: {'penalty': 'l1', 'C': 2},

 mean: 0.90100, std: 0.01258, params: {'penalty': 'l2', 'C': 2},

 mean: 0.90200, std: 0.01117, params: {'penalty': 'l1', 'C': 3},

 mean: 0.90100, std: 0.01258, params: {'penalty': 'l2', 'C': 3},

 mean: 0.90100, std: 0.01258, params: {'penalty': 'l1', 'C': 4},

 mean: 0.90100, std: 0.01258, params: {'penalty': 'l2', 'C': 4}]
```

我们可能打算获取最大得分：

```
>>> gs.grid_scores_[1][1]
0.90100000000000002

>>> max(gs.grid_scores_, key=lambda x: x[1])
mean: 0.90300, std: 0.01192, params: {'penalty': 'l1', 'C': 1}
```

获取的参数就是我们的逻辑回归的最佳选择。

5.7 使用伪造的估计器来比较结果

这个秘籍关于创建伪造的估计其。这并不是一个漂亮或有趣的东西，但是我们值得为最后构建的模型创建一个参照点。

准备

这个秘籍中，我们会执行下列任务：

1. 创建一些随机数据
2. 训练多种伪造的估计器

我们会对回归数据和分类数据来执行这两个步骤。

操作步骤

首先，我们创建随机数据：

```
>>> X, y = make_regression()

>>> from sklearn import dummy

>>> dumdum = dummy.DummyRegressor()

>>> dumdum.fit(X, y)

DummyRegressor(constant=None, strategy='mean')
```

通常，估计器仅仅使用数据的均值来做预测。

```
>>> dumdum.predict(X)[:5]

array([ 2.23297907,  2.23297907,  2.23297907,  2.23297907,
    2.23297907])
```

我们可以尝试另外两种策略。我们可以提供常数来做预测（就是上面命令中的 `constant=None` ），也可以使用中位值来预测。

如果策略是 `constant` ，才会使用提供的常数。

让我们看一看：

```
>>> predictors = [("mean", None),
                  ("median", None),
                  ("constant", 10)]

>>> for strategy, constant in predictors:
        dumdum = dummy.DummyRegressor(strategy=strategy,
                      constant=constant)

>>> dumdum.fit(X, y)

>>> print "strategy: {}".format(strategy), ",".join(map(str,
         dumdum.predict(X)[:5]))

strategy: mean 2.23297906733,2.23297906733,2.23297906733,2.23297
906733,2 .23297906733
strategy: median 20.38535248,20.38535248,20.38535248,20.38535248,
20.38535 248
strategy: constant 10.0,10.0,10.0,10.0,10.0
```

我们实际上有四种分类器的选项。这些策略类似于连续情况，但是适用于分类问题：

```
>>> predictors = [("constant", 0),
                  ("stratified", None),
                  ("uniform", None),
                  ("most_frequent", None)]
```

我们也需要创建一些分类数据：

```
>>> X, y = make_classification()
>>> for strategy, constant in predictors:
        dumdum = dummy.DummyClassifier(strategy=strategy,
                      constant=constant)
        dumdum.fit(X, y)
        print "strategy: {}".format(strategy), ",".join(map(str,
              dumdum.predict(X)[:5]))
strategy: constant 0,0,0,0,0
strategy: stratified 1,0,0,1,0
strategy: uniform 0,0,0,1,1
strategy: most_frequent 1,1,1,1,1
```

工作原理

最好在最简单的模型上测试你的模型，这就是伪造的估计器的作用。例如，在一个模型中，5% 的数据是伪造的。所以，我们可能能够训练出一个漂亮的模型，而不需要猜测任何伪造。

我们可以通过使用分层（ `stratified` ）策略来床架买模型，使用下面的命令。我们也可以获取一个不错的示例，关于为什么分类的不均等会导致问题：

```
>>> X, y = make_classification(20000, weights=[.95, .05])

>>> dumdum = dummy.DummyClassifier(strategy='most_frequent')

>>> dumdum.fit(X, y)
DummyClassifier(constant=None, random_state=None, strategy='most
_ frequent')

>>> from sklearn.metrics import accuracy_score

>>> print accuracy_score(y, dumdum.predict(X))

0.94575
```

我们实际上经常是正确的，但关键不是这个。关键是，这就是我们的基线。如果我们不能为伪造数据创建模型，并且比这个更准确，它就不值得我们花时间。

5.8 回归模型评估

我们已经学过了如何量化分类中的误差，现在我们讨论连续问题中的误差。例如，我们尝试预测年龄而不是性别。

准备

像分类一样，我们伪造一些数据，之后绘制变化。我们开始会很简单，之后逐步变复杂。数据是模拟的线性模型。

```
m = 2
b = 1
y = lambda x: m * x + b
```

同时，导入我们的模块：

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn import metrics
```

操作步骤

我们会执行下列操作：

1. 使用 `y` 来生成 `y_actual`
2. 使用 `y_actual` 加上一些 `err` 生成 `y_prediction'`
3. 绘制差异
4. 遍历不同的度量并绘制它们

让我们同时关注步骤 1 和 2，并且创建一个函数来帮助我们。这与我们刚刚看的相同，但是我们添加一些功能来指定误差（如果是个常量则为偏差）。

```
>>> def data(x, m=2, b=1, e=None, s=10):
        """
        Args:
            x: The x value
            m: Slope
            b: Intercept
            e: Error, optional, True will give random error

        """

        if e is None:
            e_i = 0
        elif e is True:
            e_i = np.random.normal(0, s, len(xs))
        else:
            e_i = e

        return x * m + b + e_i
```

既然我们已经拥有了函数，让我们定义 `y_hat` 和 `y_actual` 。我们会以便利的方法来实现：

```
>>> from functools import partial

>>> N = 100
>>> xs = np.sort(np.random.rand(N)*100)

>>> y_pred_gen = partial(data, x=xs, e=True)
>>> y_true_gen = partial(data, x=xs)

>>> y_pred = y_pred_gen()
>>> y_true = y_true_gen()

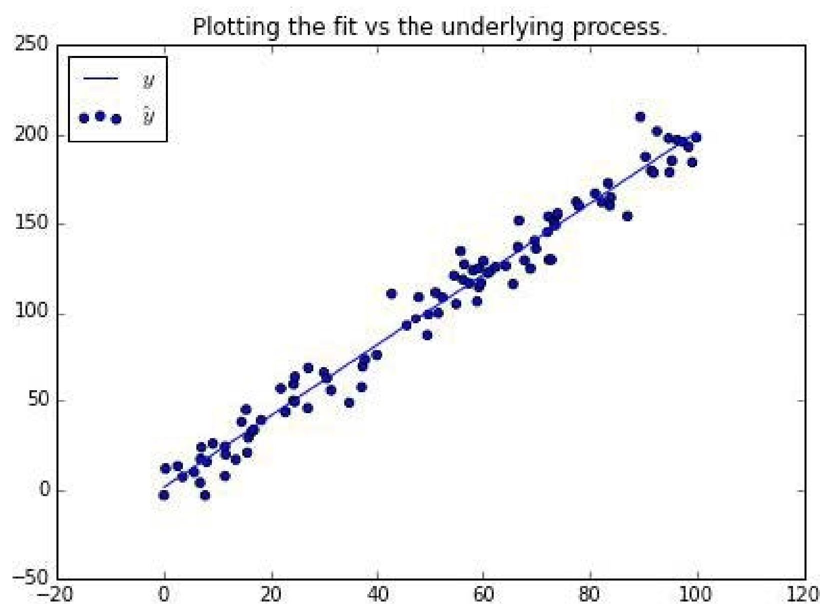
>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Plotting the fit vs the underlying process.")
>>> ax.scatter(xs, y_pred, label=r'$\hat{y}$')

>>> ax.plot(xs, y_true, label=r'$y$')

>>> ax.legend(loc='best')
```

输出如下：



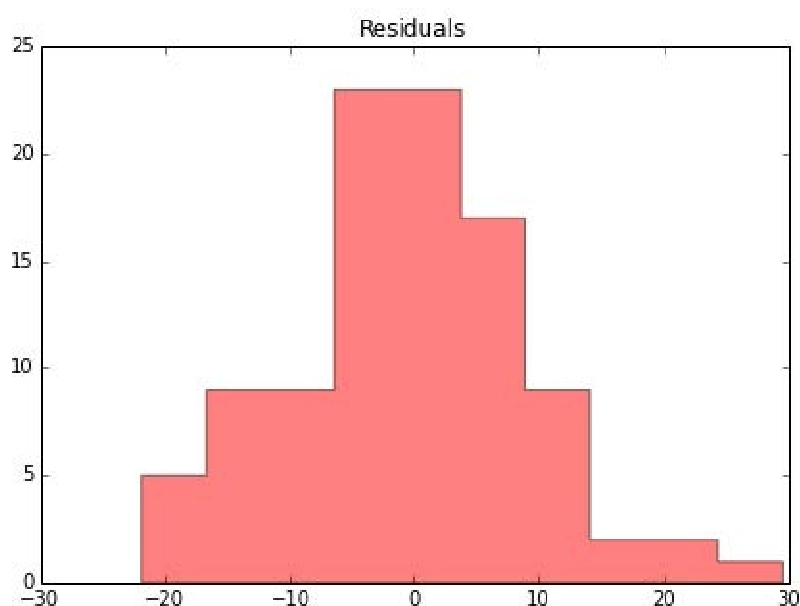
仅仅为了验证输出，我们要计算经典的残差。

```
>>> e_hat = y_pred - y_true

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.set_title("Residuals")
>>> ax.hist(e_hat, color='r', alpha=.5, histtype='stepfilled')
```

输出如下：



看起来不错。

工作原理

现在让我们看看度量。

首先，一种度量就是均方误差。

```
MSE(y_trus, y_pred) = E((y_trus - y_pred)^2)
```

```
mse = ((y_trus - y_pred) ** 2).mean()
```

你可以使用下面的代码来计算均方误差值：

```
>>> metrics.mean_squared_error(y_true, y_pred)

93.342352628475368
```

要注意，这个代码会惩罚更大误差。要注意，我们这里所做的是，将模型的可能的损失函数应用于测试数据。

另一个萱萱个就是平均绝对差。我们需要计算差异的绝对值。如果我们不这么做，我们的值就可能接近于零，也就是分布的均值：

```
MAD(y_trus, y_pred) = E(|y_trus - y_pred|)
```

```
mad = np.abs(y_trus - y_pred).mean()
```

最终的选项是 R 平方，它是 1 减去拟合模型的均方误差，与整体均值的军方误差的比值。随着比值接近于 0，R 平方接近于 1。

```
rsq = 1 - ((y_trus - y_pred) ** 2).sum() / ((y_trus - y_trus.mea
n()) ** 2).sum()
```

```
>>> metrics.r2_score(y_true, y_pred)

0.9729312117010761
```

R 平方是描述性的，它不提供模型准确性的清晰感觉。

5.9 特征选取

这个秘籍以及后面那个都关于自动特征选取。我喜欢将其看做参数调整的特征替换。就像我们做交叉验证来寻找合适的通用参数，我们可以寻找合适的特征通用子集。这涉及到几种不同方式。

最简单的想法就是到那边了选取。其它方法涉及到处理特征的组合。

特征选取的一个额外好处就是，它可以减轻数据收集的负担。想象你已经在一个很小的数据子集上构建了模型。如果一切都很好，你可能打算扩展来预测数据的整个子集。如果是这样，你可以减少数据收集的工作量。

准备

在单变量选取中，评分函数又出现了。这次，它们会定义比较度量，我们可以用它来去掉一些特征。

这个秘籍中，我们会训练带有 10000 个特征的回归模型，但是只有 1000 个点。我们会浏览多种单变量特征选取方式。

```
>>> from sklearn import datasets
>>> X, y = datasets.make_regression(1000, 10000)
```

既然我们拥有了数据，我们会使用多种方式来比较特征。当你进行文本分析，或者一些生物信息学分析时，这是个非常常见的情况。

操作步骤

首先，我们需要导入 `feature_selection` 模块。

```
>>> from sklearn import feature_selection
>>> f, p = feature_selection.f_regression(X, y)
```

这里， `f` 就是和每个线性模型的特征之一相关的 f 分数。我们之后可以比较这些特征，并基于这个比较，我们可以筛选特征。 `p` 是 `f` 值对应的 p 值。

在统计学中， `p` 值是一个值的概率，它比检验统计量的当前值更极端。这里 `f` 值检验统计量。

```
>>> f[:5]
array([  1.06271357e-03, 2.91136869e+00, 1.01886922e+00,
         2.22483130e+00, 4.67624756e-01])
>>> p[:5]
array([ 0.97400066, 0.08826831, 0.31303204, 0.1361235, 0.49424067
])
```

我们可以看到，许多 `p` 值都太大了。我们更想让 `p` 值变小。所以我们可以将 NumPy 从工具箱中取出来，并且选取小于 `.05` 的 `p` 值。这些就是我们用于分析的特征。

```
>>> import numpy as np
>>> idx = np.arange(0, X.shape[1])
>>> features_to_keep = idx[p < .05]
>>> len(features_to_keep)
501
```

你可以看到，我们实际上保留了相当大的特征总量。取决于模型的上下文，我们可以减少 `p` 至。这会减少保留的特征数量。

另一个选择是使用 `VarianceThreshold` 对象。我们已经了解一些了。但是重要的是理解，我们训练模型的能力，基本上是基于特征所产生的变化。如果没有变化，我们的特征就不能描述独立变量的变化。根据文档，良好的特征可以用于非监督案例，因为它并不是结果变量。

我们需要设置起始值来筛选特征。为此，我们选取并提供特征方差的中位值。

```
>>> var_threshold = feature_selection.VarianceThreshold(np.media
n(np.
                    var(X, axis=1)))

>>> var_threshold.fit_transform(X).shape

(1000, 4835)
```

我们可以看到，我们筛选了几乎一半的特征，或多或少就是我们的预期。

工作原理

通常，所有这些方式的原理都是使用单个特征来训练基本的模型。取决于它是分类问题还是回归问题，我们可以使用合适的评分函数。

让我们观察一个更小的问题，并可视化特征选取如何筛选特定的特征。我们使用第一个示例的相同评分函数，但是仅仅有 20 个特征。

```
>>> X, y = datasets.make_regression(10000, 20)

>>> f, p = feature_selection.f_regression(X, y)
```

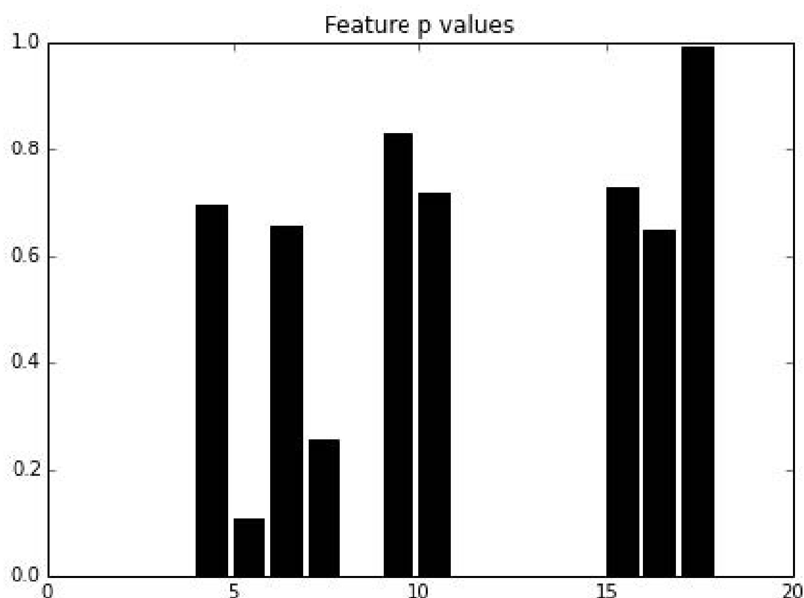
现在，让我们绘制特征的 `p` 值，我们可以看到筛选和保留哪个特征：

```
>>> from matplotlib import pyplot as plt

>>> f, ax = plt.subplots(figsize=(7, 5))

>>> ax.bar(np.arange(20), p, color='k')
>>> ax.set_title("Feature p values")
```

输出如下：



我们可以看到，许多特征没有保留，但是保留了一些特征。

5.10 L1 范数上的特征选取

我们打算实现一些相似的理念，我们在套索回归的秘籍中见过他们。在那个米几种，我们查看了含有 0 系数的特征数量。

现在我们打算更进一步，并使用 L1 范数来预处理特征。

准备

我们要使用糖尿病数据集来拟合回归。首先，我们要使用 `ShuffleSplit` 交叉验证来训练基本的 `LinearRegression` 模型，之后，我们使用 `LassoRegression` 来寻找 L1 惩罚为 0 的系数。我们希望它能帮助我们避免过拟合，也就是说这个模型非常特定于所训练的数据。换句话说，如果过拟合的话，模型并不能推广到外围的数据。

我们打算执行下列步骤：

1. 加载数据集
2. 训练基本的线性回归模型
3. 使用特征选取来移除不提供信息的特征
4. 重新训练线性回归，并与特征完整的模型相比，它拟合得多好。

操作步骤

首先加载数据集：

```
>>> import sklearn.datasets as ds
>>> diabetes = ds.load_diabetes()
```

让我们导入度量模块的 `mean_squared_error` 函数，以及 `cross_validation` 模块的 `ShuffleSplit` 交叉验证函数。

```
>>> from sklearn import metrics
>>> from sklearn import cross_validation

>>> shuff = cross_validation.ShuffleSplit(diabetes.target.size
```

现在训练模型，我们会跟踪 `ShuffleSplit` 每次迭代中的均方误差。

```
>>> mses = []
>>> for train, test in shuff:
        train_X = diabetes.data[train]
        train_y = diabetes.target[train]

        test_X = diabetes.data[~train]
        test_y = diabetes.target[~train]

        lr.fit(train_X, train_y)
        mses.append(metrics.mean_squared_error(test_y,

                    lr.predict(test_X)))

>>> np.mean(mses)
2856.366626198198
```

所以既然我们做了常规拟合，让我们在筛选系数为 0 的特征之后再检查它。让我们训练套索回归：

```
>>> from sklearn import feature_selection
>>> from sklearn import cross_validation

>>> cv = linear_model.LassoCV()
>>> cv.fit(diabetes.data, diabetes.target)
>>> cv.coef_

array([ -0. , -226.2375274 ,  526.85738059,  314.44026013,
        -196.92164002, 1.48742026, -151.78054083, 106.52846989,
        530.58541123, 64.50588257])
```

我们会移除第一个特征。我使用 NumPy 数组来表示模块中包含的列。

```
>>> import numpy as np
>>> columns = np.arange(diabetes.data.shape[1])[cv.coef_ != 0]
>>> columns array([1, 2, 3 4, 5, 6, 7, 8, 9])
```

好的，所以现在我们使用特定的特征来训练模型（请见下面代码中的列）：

```
>>> l1mses = []
>>> for train, test in shuff:
        train_X = diabetes.data[train][:, columns]
        train_y = diabetes.target[train]

        test_X = diabetes.data[~train][:, columns]
        test_y = diabetes.target[~train]

        lr.fit(train_X, train_y)           l1mses.append(metrics
.mean_squared_error(test_y,
                      lr.predict(test_X)))

>>> np.mean(l1mses)
2861.0763924492171
>>> np.mean(l1mses) - np.mean(mses)
4.7097662510191185
```

我们可以看到，即使我们移除了不提供信息的特征，模型依然不怎么样。这种情况不是始终发生。下一部分中，我们会比较模型间的拟合，其中有很多不提供信息的特征。

工作原理

首先，我们打算创建回归数据集，带有很多不提供信息的特征：

```
>>> X, y = ds.make_regression(noise=5)
```

让我们训练普通的回归：

```
>>> mses = []

>>> shuff = cross_validation.ShuffleSplit(y.size)

>>> for train, test in shuff:
        train_X = X[train]
        train_y = y[train]

        test_X = X[~train]
        test_y = y[~train]

        lr.fit(train_X, train_y)           mses.append(metrics.m
ean_squared_error(test_y,
                    lr.predict(test_X)))

>>> np.mean(mses)
879.75447864034209
```

现在我们可以以相同个过程来使用套索回归：

```
>>> cv.fit(X, y)

LassoCV(alphas=None, copy_X=True, cv=None, eps=0.001,
        fit_intercept=True, max_iter=1000, n_alphas=100,
        n_jobs=1, normalize=False, positive=False, precompute='a
uto',
        tol=0.0001, verbose=False)
```

我们会再次创建列。这是个很好的模式，让我们能够制定要包含的列。

```
>>> import numpy as np
>>> columns = np.arange(X.shape[1])[cv.coef_ != 0]
>>> columns[:5] array([11, 15, 17, 20, 21,])
>>> mses = []

>>> shuff = cross_validation.ShuffleSplit(y.size)
>>> for train, test in shuff:
        train_X = X[train][:, columns]
        train_y = y[train]

        test_X = X[~train][:, columns]
        test_y = y[~train]

        lr.fit(train_X, train_y)
        mses.append(metrics.mean_squared_error(test_y,

                        lr.predict(test_X)))

>>> np.mean(mses)
15.755403220117708
```

我们可以看到，我们在模型的训练中获得了极大的提升。这刚好解释了，我们需要承认，不是所有特征都需要或者应该放进模型中。

5.11 使用 joblib 保存模型

这个秘籍中，我们打算展示如何保存模型，便于以后使用。例如，你可能打算实际使用模型来预测结果，并自动做出决策。

准备

这个秘籍中，我们会执行下列任务：

1. 训练我们要保存的模型
2. 导入 joblib 并保存模型

操作步骤

为了使用 joblib 保存我们的模型，可以使用下面的代码：

```
>>> from sklearn import datasets, tree
>>> X, y = datasets.make_classification()

>>> dt = tree.DecisionTreeClassifier()
>>> dt.fit(X, y)

DecisionTreeClassifier(compute_importances=None, criterion='gini'
,
                       max_depth=None, max_features=None,
                       max_leaf_nodes=None, min_density=None,
                       min_samples_leaf=1, min_samples_split=2,
                       random_state=None, splitter='best')

>>> from sklearn.externals import joblib
>>> joblib.dump(dt, "dtree.clf")
['dtree.clf',
 'dtree.clf_01.npy',
 'dtree.clf_02.npy',
 'dtree.clf_03.npy',
 'dtree.clf_04.npy']
```

工作原理

上面的下面的原理是，保存对象状态，可以重新加载进 Sklearn 对象。要注意，对于不同的模型类型，模型的状态拥有不同的复杂度级别。

出于简单的因素，将我们要保存的东西看做一种方式，我们提供出入来预测结果。对于回归来说很简单，简单的线性代数就足以。但是，对于像是随机森林的模型，我们可能拥有很多颗树。这些树可能拥有不同的复杂度级别，比较困难。

更多

我们可以简单随机森林模型的大小：

```
>>> from sklearn import ensemble

>>> rf = ensemble.RandomForestClassifier()
>>> rf.fit(X, y)

RandomForestClassifier(bootstrap=True, compute_importances=None,
                        criterion='gini', max_depth=None,
                        max_features='auto', max_leaf_nodes=None
,
                        min_density=None, min_samples_leaf=1,
                        min_samples_split=2, n_estimators=10,
                        n_jobs=1, oob_score=False,
                        random_state=None, verbose=0)
```

我打算省略输出，但是总之，我的机器上一共有 52 个输出文件。

```
>>> joblib.dump(rf, "rf.clf")
['rf.clf',
 'rf.clf_01.npy',
 'rf.clf_02.npy',
 'rf.clf_03.npy',
 'rf.clf_04.npy',
 'rf.clf_05.npy',
 'rf.clf_06.npy',
 ...             ]
```